System V Interface Definition
System V
Interface Definition
Issue 2

Volume I
Copyright © 1986 AT&T. All Rights Reserved.

No part of this publication may be reproduced or transmitted in any form or by any means — graphic, electronic, electrical, mechanical, or chemical, including photocopying, recording in any medium, taping, by any computer or information storage and retrieval systems, etc., without prior permission in writing from AT&T.

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 or 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 of 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 trademark of AT&T.
AP5-S 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)

You must use a major credit card for orders made by phone.

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

AT&T Customer Information Center (CIC)
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
307-127         All Volumes
# Table of Contents

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
</tr>
</tbody>
</table>

## Part I

### A General Introduction to the System V Interface Definition

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>General Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Future Directions</td>
</tr>
</tbody>
</table>

## Part II

### Base System Definition

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Introduction</td>
</tr>
<tr>
<td>4</td>
<td>Definitions</td>
</tr>
<tr>
<td>5</td>
<td>Environment</td>
</tr>
<tr>
<td>6</td>
<td>OS Service Routines</td>
</tr>
<tr>
<td>7</td>
<td>General Library Routines</td>
</tr>
</tbody>
</table>

## Part III

### Kernel Extension Definition

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Introduction</td>
</tr>
<tr>
<td>9</td>
<td>Definitions</td>
</tr>
<tr>
<td>10</td>
<td>Environment</td>
</tr>
<tr>
<td>11</td>
<td>OS Service Routines</td>
</tr>
</tbody>
</table>

### Appendix

#### Changes from Issue 1

| Base System Definition | 298 |
| Kernel Extension Differences | 303 |

### Indexes

<table>
<thead>
<tr>
<th>Volume I</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Index</td>
</tr>
<tr>
<td>Function Index</td>
</tr>
</tbody>
</table>
The *System V Interface Definition* 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 runtime 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 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 issue of the System V Interface Definition corresponds to functionality in AT&T System V Release 1.0 and System V Release 2.0. An implementation of System V may conform to the System V Release 1.0 functionality or the System V Release 2.0 functionality. All System V Release 2.0 enhancements to System V Release 1.0 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

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 \(malloc\) points to the MALLOC(BA_OS) pages, because the function \(malloc\) 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 two references define the C language for System V Release 1.0 and System V Release 2.0 respectively:

Chapter 2
Future Directions

2.1 NETWORK SERVICES EXTENSION

The Network Services Extension will provide advanced standard interfaces to support networking applications. It is divided into three functional areas. The Open Systems Networking Interfaces section describes a protocol-independent application interface to transport services based on the Open Systems Interconnection (OSI) Reference Model [IS 7498]. The Streams I/O Interfaces section describes the operating system service routines that provide direct access to protocol modules implemented using the streams framework. The Shared Resource Environment section describes new capabilities for sharing and administering resources among interconnected machines.

2.2 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.

2.3 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.

2.4 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.
2.5 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).

2.6 TERMINAL INTERFACE EXTENSION
The current Terminal Interface Extension 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 terminal actually in use. This Extension will be enhanced to support applications on both character and bit-mapped terminals and to provide capabilities for handling windows, menus, icons, etc. which can be accessed by a keyboard or other input device, such as a mouse. Applications written in this environment will have a uniform and easily used human interface. In addition, applications which rely on curses/terminfo will be compatible with the new environment.

2.7 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:

<table>
<thead>
<tr>
<th>Code-Set</th>
<th>Internal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0 (Primary code-set)</td>
<td>0XXXXXXX</td>
</tr>
<tr>
<td>Set 1 (Supplementary code-set #1)</td>
<td>1XXXXXXX 1XXXXXXX</td>
</tr>
<tr>
<td>Set 2 (Supplementary code-set #2)</td>
<td>SS2 1XXXXXXX 1XXXXXXX</td>
</tr>
<tr>
<td>Set 3 (Supplementary code-set #3)</td>
<td>SS3 1XXXXXXX 1XXXXXXX</td>
</tr>
</tbody>
</table>

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.

General Introduction
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, JIS 6226, 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, 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.

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 an 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.
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
Chapter 3
Introduction

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. 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 Operating System Service Routines whose run-time behavior must be supported by any implementation of the Base System.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>ABORT(BA_OS)</td>
</tr>
<tr>
<td>access</td>
<td>ACCESS(BA_OS)</td>
</tr>
<tr>
<td>alarm</td>
<td>ALARM(BA_OS)</td>
</tr>
<tr>
<td>calloc</td>
<td>MALLOC(BA_OS)</td>
</tr>
<tr>
<td>chdir</td>
<td>CHDIR(BA_OS)</td>
</tr>
<tr>
<td>chmod</td>
<td>CHMOD(BA_OS)</td>
</tr>
<tr>
<td>chown</td>
<td>CHOWN(BA_OS)</td>
</tr>
<tr>
<td>clearerr</td>
<td>FERROR(BA_OS)</td>
</tr>
<tr>
<td>dup</td>
<td>DUP(BA_OS)</td>
</tr>
<tr>
<td>exit</td>
<td>EXIT(BA_OS)</td>
</tr>
<tr>
<td>fclose</td>
<td>FCLOSE(BA_OS)</td>
</tr>
<tr>
<td>fcntl</td>
<td>FCNTL(BA_OS)</td>
</tr>
<tr>
<td>fdopen</td>
<td>FOPEN(BA_OS)</td>
</tr>
<tr>
<td>feof</td>
<td>FERROR(BA_OS)</td>
</tr>
<tr>
<td>ferror</td>
<td>FERROR(BA_OS)</td>
</tr>
<tr>
<td>fflush</td>
<td>FCLOSE(BA_OS)</td>
</tr>
<tr>
<td>fileno</td>
<td>FERROR(BA_OS)</td>
</tr>
<tr>
<td>fopen</td>
<td>FOPEN(BA_OS)</td>
</tr>
<tr>
<td>free</td>
<td>MALLOC(BA_OS)</td>
</tr>
<tr>
<td>freopen</td>
<td>FOPEN(BA_OS)</td>
</tr>
<tr>
<td>fseek</td>
<td>FSEEK(BA_OS)</td>
</tr>
<tr>
<td>fstat</td>
<td>STAT(BA_OS)</td>
</tr>
<tr>
<td>ftell</td>
<td>FSEEK(BA_OS)</td>
</tr>
<tr>
<td>fwrite</td>
<td>FREAD(BA_OS)</td>
</tr>
<tr>
<td>getcwd</td>
<td>GETCWD(BA_OS)</td>
</tr>
<tr>
<td>getegid</td>
<td>GETUID(BA_OS)</td>
</tr>
<tr>
<td>geteuid</td>
<td>GETUID(BA_OS)</td>
</tr>
<tr>
<td>getgid</td>
<td>GETUID(BA_OS)</td>
</tr>
<tr>
<td>getpgrp</td>
<td>GETPGRP(BA_OS)</td>
</tr>
<tr>
<td>getppid</td>
<td>GETPPID(BA_OS)</td>
</tr>
<tr>
<td>close</td>
<td>CLOSE(BA_OS)</td>
</tr>
<tr>
<td>creat</td>
<td>CREATE(BA_OS)</td>
</tr>
<tr>
<td>exec</td>
<td>EXEC(BA_OS)</td>
</tr>
<tr>
<td>execl</td>
<td>EXEC(BA_OS)</td>
</tr>
<tr>
<td>execlp</td>
<td>EXEC(BA_OS)</td>
</tr>
<tr>
<td>execv</td>
<td>EXEC(BA_OS)</td>
</tr>
<tr>
<td>execvp</td>
<td>EXEC(BA_OS)</td>
</tr>
<tr>
<td>_exit</td>
<td>EXIT(BA_OS)</td>
</tr>
<tr>
<td>fork</td>
<td>FORK(BA_OS)</td>
</tr>
<tr>
<td>lseek</td>
<td>LSEEK(BA_OS)</td>
</tr>
<tr>
<td>mount</td>
<td>MOUNT(BA_OS)</td>
</tr>
<tr>
<td>open</td>
<td>OPEN(BA_OS)</td>
</tr>
<tr>
<td>read</td>
<td>READ(BA_OS)</td>
</tr>
<tr>
<td>umount</td>
<td>U Mount(BA_OS)</td>
</tr>
<tr>
<td>write</td>
<td>WRITE(BA_OS)</td>
</tr>
</tbody>
</table>

**TABLE 3-1. Base System: OS Service Routines**
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, Part III — Kernel Extension.

All the routines in Table 3-1, except those marked with † or ††, are common to System V Release 1.0 and System V Release 2.0. 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.

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>
<thead>
<tr>
<th>Table 3-2. Base System: General Library Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>abs</strong></td>
</tr>
<tr>
<td><strong>acos</strong></td>
</tr>
<tr>
<td><strong>asin</strong></td>
</tr>
<tr>
<td><strong>atan2</strong></td>
</tr>
<tr>
<td><strong>atan</strong></td>
</tr>
<tr>
<td><strong>ceil</strong></td>
</tr>
<tr>
<td><strong>cos</strong></td>
</tr>
<tr>
<td><strong>cosh</strong></td>
</tr>
<tr>
<td><strong>erf</strong></td>
</tr>
<tr>
<td><strong>erfc</strong></td>
</tr>
<tr>
<td><strong>exp</strong></td>
</tr>
<tr>
<td><strong>fabs</strong></td>
</tr>
<tr>
<td><strong>floor</strong></td>
</tr>
<tr>
<td><strong>fmod</strong></td>
</tr>
<tr>
<td><strong>frexp</strong></td>
</tr>
<tr>
<td><strong>gamma</strong></td>
</tr>
<tr>
<td><strong>hypot</strong></td>
</tr>
<tr>
<td><strong>tolower</strong></td>
</tr>
<tr>
<td><strong>toupper</strong></td>
</tr>
<tr>
<td><strong>advance</strong></td>
</tr>
<tr>
<td><strong>asctime</strong></td>
</tr>
<tr>
<td><strong>atof</strong></td>
</tr>
<tr>
<td><strong>atoi</strong></td>
</tr>
<tr>
<td><strong>compile</strong></td>
</tr>
<tr>
<td><strong>crypt</strong></td>
</tr>
<tr>
<td><strong>ctime</strong></td>
</tr>
<tr>
<td><strong>encrypt</strong></td>
</tr>
<tr>
<td><strong>gmtime</strong></td>
</tr>
<tr>
<td><strong>isalnum</strong></td>
</tr>
<tr>
<td><strong>isalpha</strong></td>
</tr>
<tr>
<td><strong>isascii</strong></td>
</tr>
<tr>
<td><strong>iscntrl</strong></td>
</tr>
<tr>
<td><strong>isdigit</strong></td>
</tr>
<tr>
<td><strong>isgraph</strong></td>
</tr>
<tr>
<td><strong>islower</strong></td>
</tr>
<tr>
<td><strong>isprint</strong></td>
</tr>
<tr>
<td><strong>ispunct</strong></td>
</tr>
<tr>
<td><strong>isspace</strong></td>
</tr>
<tr>
<td><strong>isupper</strong></td>
</tr>
<tr>
<td><strong>isxdigit</strong></td>
</tr>
<tr>
<td><strong>localtime</strong></td>
</tr>
<tr>
<td><strong>tzset</strong></td>
</tr>
<tr>
<td>Function</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>bsearch</td>
</tr>
<tr>
<td>clock</td>
</tr>
<tr>
<td>ctermid</td>
</tr>
<tr>
<td>drand48</td>
</tr>
<tr>
<td>erand48</td>
</tr>
<tr>
<td>fgetc</td>
</tr>
<tr>
<td>fgets</td>
</tr>
<tr>
<td>fprintf</td>
</tr>
<tr>
<td>putc</td>
</tr>
<tr>
<td>putenv</td>
</tr>
<tr>
<td>qsort</td>
</tr>
<tr>
<td>rand</td>
</tr>
<tr>
<td>scanf</td>
</tr>
<tr>
<td>putw</td>
</tr>
<tr>
<td>ftw</td>
</tr>
<tr>
<td>getc</td>
</tr>
<tr>
<td>getchar</td>
</tr>
<tr>
<td>getenv</td>
</tr>
<tr>
<td>getopt</td>
</tr>
<tr>
<td>gets</td>
</tr>
<tr>
<td>getw</td>
</tr>
<tr>
<td>gsignal</td>
</tr>
<tr>
<td>hcreate</td>
</tr>
<tr>
<td>hdestroy</td>
</tr>
<tr>
<td>hsearch</td>
</tr>
<tr>
<td>isatty</td>
</tr>
<tr>
<td>jrand48</td>
</tr>
<tr>
<td>lcong48</td>
</tr>
<tr>
<td>lfind</td>
</tr>
<tr>
<td>longjmp</td>
</tr>
<tr>
<td>rand48</td>
</tr>
<tr>
<td>lsearch</td>
</tr>
<tr>
<td>mktemp</td>
</tr>
<tr>
<td>mrand48</td>
</tr>
<tr>
<td>nrand48</td>
</tr>
</tbody>
</table>

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 † are in System V Release 2.0 only, while all others are in both System V Release 1.0 and System V Release 2.0. 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.
ASCII character set

Tables 3-1 and 3-2 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>
<thead>
<tr>
<th>Octal</th>
<th>Character</th>
<th>Hexadecimal</th>
<th>Octal</th>
<th>Character</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>nul</td>
<td>0</td>
<td>0</td>
<td>bel</td>
<td>\x07</td>
</tr>
<tr>
<td>1</td>
<td>soh</td>
<td>0x01</td>
<td>1</td>
<td>ack</td>
<td>\x04</td>
</tr>
<tr>
<td>2</td>
<td>stx</td>
<td>0x02</td>
<td>2</td>
<td>syn</td>
<td>\x06</td>
</tr>
<tr>
<td>3</td>
<td>etx</td>
<td>0x03</td>
<td>3</td>
<td>etb</td>
<td>\x02</td>
</tr>
<tr>
<td>4</td>
<td>eot</td>
<td>0x04</td>
<td>4</td>
<td>0</td>
<td>\x00</td>
</tr>
<tr>
<td>5</td>
<td>enq</td>
<td>0x05</td>
<td>5</td>
<td>0</td>
<td>\x06</td>
</tr>
<tr>
<td>6</td>
<td>ack</td>
<td>0x06</td>
<td>6</td>
<td>0</td>
<td>\x07</td>
</tr>
<tr>
<td>7</td>
<td>bel</td>
<td>0x07</td>
<td>7</td>
<td>0</td>
<td>\x08</td>
</tr>
<tr>
<td>10</td>
<td>bs</td>
<td>0x0A</td>
<td>10</td>
<td>0</td>
<td>\x0B</td>
</tr>
<tr>
<td>11</td>
<td>so</td>
<td>0x0B</td>
<td>11</td>
<td>0</td>
<td>\x0C</td>
</tr>
<tr>
<td>12</td>
<td>st</td>
<td>0x0C</td>
<td>12</td>
<td>0</td>
<td>\x0D</td>
</tr>
<tr>
<td>13</td>
<td>et</td>
<td>0x0D</td>
<td>13</td>
<td>0</td>
<td>\x0E</td>
</tr>
<tr>
<td>14</td>
<td>e</td>
<td>0x0E</td>
<td>14</td>
<td>0</td>
<td>\x0F</td>
</tr>
<tr>
<td>15</td>
<td>en</td>
<td>0x10</td>
<td>15</td>
<td>0</td>
<td>0x00</td>
</tr>
<tr>
<td>16</td>
<td>ack</td>
<td>0x11</td>
<td>16</td>
<td>0</td>
<td>0x01</td>
</tr>
<tr>
<td>17</td>
<td>bel</td>
<td>0x12</td>
<td>17</td>
<td>0</td>
<td>0x02</td>
</tr>
<tr>
<td>18</td>
<td>bs</td>
<td>0x13</td>
<td>18</td>
<td>0</td>
<td>0x03</td>
</tr>
<tr>
<td>19</td>
<td>so</td>
<td>0x14</td>
<td>19</td>
<td>0</td>
<td>0x04</td>
</tr>
<tr>
<td>20</td>
<td>st</td>
<td>0x15</td>
<td>20</td>
<td>0</td>
<td>0x05</td>
</tr>
<tr>
<td>21</td>
<td>et</td>
<td>0x16</td>
<td>21</td>
<td>0</td>
<td>0x06</td>
</tr>
<tr>
<td>22</td>
<td>e</td>
<td>0x17</td>
<td>22</td>
<td>0</td>
<td>0x07</td>
</tr>
<tr>
<td>23</td>
<td>en</td>
<td>0x18</td>
<td>23</td>
<td>0</td>
<td>0x08</td>
</tr>
<tr>
<td>24</td>
<td>ack</td>
<td>0x19</td>
<td>24</td>
<td>0</td>
<td>0x09</td>
</tr>
<tr>
<td>25</td>
<td>bel</td>
<td>0x1A</td>
<td>25</td>
<td>0</td>
<td>0x0A</td>
</tr>
<tr>
<td>26</td>
<td>bs</td>
<td>0x1B</td>
<td>26</td>
<td>0</td>
<td>0x0B</td>
</tr>
<tr>
<td>27</td>
<td>so</td>
<td>0x1C</td>
<td>27</td>
<td>0</td>
<td>0x0C</td>
</tr>
<tr>
<td>28</td>
<td>st</td>
<td>0x1D</td>
<td>28</td>
<td>0</td>
<td>0x0D</td>
</tr>
<tr>
<td>29</td>
<td>et</td>
<td>0x1E</td>
<td>29</td>
<td>0</td>
<td>0x0E</td>
</tr>
<tr>
<td>30</td>
<td>e</td>
<td>0x1F</td>
<td>30</td>
<td>0</td>
<td>0x0F</td>
</tr>
<tr>
<td>31</td>
<td>en</td>
<td>0x100</td>
<td>31</td>
<td>0</td>
<td>0x20</td>
</tr>
<tr>
<td>32</td>
<td>ack</td>
<td>0x101</td>
<td>32</td>
<td>0</td>
<td>0x21</td>
</tr>
<tr>
<td>33</td>
<td>bel</td>
<td>0x102</td>
<td>33</td>
<td>0</td>
<td>0x22</td>
</tr>
<tr>
<td>34</td>
<td>bs</td>
<td>0x103</td>
<td>34</td>
<td>0</td>
<td>0x23</td>
</tr>
<tr>
<td>35</td>
<td>so</td>
<td>0x104</td>
<td>35</td>
<td>0</td>
<td>0x24</td>
</tr>
<tr>
<td>36</td>
<td>st</td>
<td>0x105</td>
<td>36</td>
<td>0</td>
<td>0x25</td>
</tr>
<tr>
<td>37</td>
<td>et</td>
<td>0x106</td>
<td>37</td>
<td>0</td>
<td>0x26</td>
</tr>
<tr>
<td>38</td>
<td>e</td>
<td>0x107</td>
<td>38</td>
<td>0</td>
<td>0x27</td>
</tr>
<tr>
<td>39</td>
<td>en</td>
<td>0x108</td>
<td>39</td>
<td>0</td>
<td>0x28</td>
</tr>
</tbody>
</table>

TABLE 3-1. Octal map of ASCII character set.

<table>
<thead>
<tr>
<th>Octal</th>
<th>Character</th>
<th>Hexadecimal</th>
<th>Octal</th>
<th>Character</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>nul</td>
<td>0</td>
<td>01</td>
<td>soh</td>
<td>0x01</td>
</tr>
<tr>
<td>02</td>
<td>stx</td>
<td>0x02</td>
<td>03</td>
<td>etx</td>
<td>0x03</td>
</tr>
<tr>
<td>04</td>
<td>eot</td>
<td>0x04</td>
<td>05</td>
<td>enq</td>
<td>0x05</td>
</tr>
<tr>
<td>06</td>
<td>ack</td>
<td>0x06</td>
<td>07</td>
<td>bel</td>
<td>0x07</td>
</tr>
<tr>
<td>08</td>
<td>bs</td>
<td>0x08</td>
<td>09</td>
<td>so</td>
<td>0x09</td>
</tr>
<tr>
<td>0a</td>
<td>st</td>
<td>0x0A</td>
<td>0b</td>
<td>et</td>
<td>0x0B</td>
</tr>
<tr>
<td>0c</td>
<td>e</td>
<td>0x0C</td>
<td>0d</td>
<td>en</td>
<td>0x0D</td>
</tr>
<tr>
<td>0e</td>
<td>ack</td>
<td>0x0E</td>
<td>0f</td>
<td>bel</td>
<td>0x0F</td>
</tr>
<tr>
<td>10</td>
<td>bs</td>
<td>0x10</td>
<td>11</td>
<td>so</td>
<td>0x11</td>
</tr>
<tr>
<td>12</td>
<td>st</td>
<td>0x12</td>
<td>13</td>
<td>et</td>
<td>0x13</td>
</tr>
<tr>
<td>14</td>
<td>e</td>
<td>0x14</td>
<td>15</td>
<td>en</td>
<td>0x15</td>
</tr>
<tr>
<td>16</td>
<td>ack</td>
<td>0x16</td>
<td>17</td>
<td>bel</td>
<td>0x17</td>
</tr>
<tr>
<td>18</td>
<td>bs</td>
<td>0x18</td>
<td>19</td>
<td>so</td>
<td>0x19</td>
</tr>
<tr>
<td>1a</td>
<td>st</td>
<td>0x1A</td>
<td>1b</td>
<td>et</td>
<td>0x1B</td>
</tr>
<tr>
<td>1c</td>
<td>e</td>
<td>0x1C</td>
<td>1d</td>
<td>en</td>
<td>0x1D</td>
</tr>
<tr>
<td>1e</td>
<td>ack</td>
<td>0x1E</td>
<td>1f</td>
<td>bel</td>
<td>0x1F</td>
</tr>
<tr>
<td>20</td>
<td>bs</td>
<td>0x20</td>
<td>21</td>
<td>so</td>
<td>0x21</td>
</tr>
<tr>
<td>22</td>
<td>st</td>
<td>0x22</td>
<td>23</td>
<td>et</td>
<td>0x23</td>
</tr>
<tr>
<td>24</td>
<td>e</td>
<td>0x24</td>
<td>25</td>
<td>en</td>
<td>0x25</td>
</tr>
<tr>
<td>26</td>
<td>ack</td>
<td>0x26</td>
<td>27</td>
<td>bel</td>
<td>0x27</td>
</tr>
<tr>
<td>28</td>
<td>bs</td>
<td>0x28</td>
<td>29</td>
<td>so</td>
<td>0x29</td>
</tr>
<tr>
<td>2a</td>
<td>st</td>
<td>0x2A</td>
<td>2b</td>
<td>et</td>
<td>0x2B</td>
</tr>
<tr>
<td>2c</td>
<td>e</td>
<td>0x2C</td>
<td>2d</td>
<td>en</td>
<td>0x2D</td>
</tr>
<tr>
<td>2e</td>
<td>ack</td>
<td>0x2E</td>
<td>2f</td>
<td>bel</td>
<td>0x2F</td>
</tr>
<tr>
<td>30</td>
<td>bs</td>
<td>0x30</td>
<td>31</td>
<td>so</td>
<td>0x31</td>
</tr>
<tr>
<td>32</td>
<td>st</td>
<td>0x32</td>
<td>33</td>
<td>et</td>
<td>0x33</td>
</tr>
<tr>
<td>34</td>
<td>e</td>
<td>0x34</td>
<td>35</td>
<td>en</td>
<td>0x35</td>
</tr>
<tr>
<td>36</td>
<td>ack</td>
<td>0x36</td>
<td>37</td>
<td>bel</td>
<td>0x37</td>
</tr>
<tr>
<td>38</td>
<td>bs</td>
<td>0x38</td>
<td>39</td>
<td>so</td>
<td>0x39</td>
</tr>
<tr>
<td>3a</td>
<td>st</td>
<td>0x3A</td>
<td>3b</td>
<td>et</td>
<td>0x3B</td>
</tr>
<tr>
<td>3c</td>
<td>e</td>
<td>0x3C</td>
<td>3d</td>
<td>en</td>
<td>0x3D</td>
</tr>
<tr>
<td>3e</td>
<td>ack</td>
<td>0x3E</td>
<td>3f</td>
<td>bel</td>
<td>0x3F</td>
</tr>
</tbody>
</table>

TABLE 3-2. Hexadecimal map of ASCII character set.
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 CREATE(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\}. 

Page 28
The following lists the implementation-specific constants that may be used in System V component definitions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARG_MAX</td>
<td>max. length of argument to exec</td>
</tr>
<tr>
<td>CHAR_BIT</td>
<td>number of bits in a char</td>
</tr>
<tr>
<td>CHAR_MAX</td>
<td>max. integer value of a char</td>
</tr>
<tr>
<td>CHILD_MAX</td>
<td>max. number of processes per user-ID</td>
</tr>
<tr>
<td>CLK_TCK</td>
<td>number of clock ticks per second</td>
</tr>
<tr>
<td>FCHR_MAX</td>
<td>max. size of a file in bytes</td>
</tr>
<tr>
<td>INT_MAX</td>
<td>max. decimal value of an int</td>
</tr>
<tr>
<td>LINK_MAX</td>
<td>max. number of links to a single file</td>
</tr>
<tr>
<td>LOCK_MAX</td>
<td>max. number of entries in system lock table</td>
</tr>
<tr>
<td>LONG_BIT</td>
<td>number of bits in a long</td>
</tr>
<tr>
<td>LONG_MAX</td>
<td>max. decimal value of a long</td>
</tr>
<tr>
<td>MAX_DOUBLE</td>
<td>max. decimal value of a double</td>
</tr>
<tr>
<td>MAX_CHAR</td>
<td>max. size of character input buffer</td>
</tr>
<tr>
<td>NAME_MAX</td>
<td>max. number of characters in a file-name</td>
</tr>
<tr>
<td>OPEN_MAX</td>
<td>max. number of files a process can have open</td>
</tr>
<tr>
<td>PASS_MAX</td>
<td>max. number of significant characters in a password</td>
</tr>
<tr>
<td>PATH_MAX</td>
<td>max. number of characters in a path-name</td>
</tr>
<tr>
<td>PID_MAX</td>
<td>max. value for a process-ID</td>
</tr>
<tr>
<td>PIPE_BUF</td>
<td>max. number bytes atomic in write to a pipe</td>
</tr>
<tr>
<td>PIPE_MAX</td>
<td>max. number of bytes written to a pipe in a write</td>
</tr>
<tr>
<td>PROC_MAX</td>
<td>max. number of simultaneous processes, system wide</td>
</tr>
<tr>
<td>SHRT_MAX</td>
<td>max. decimal value of a short</td>
</tr>
<tr>
<td>STD_BLK</td>
<td>number of bytes in a physical I/O block</td>
</tr>
<tr>
<td>SYS_NMLN</td>
<td>number of characters in string returned by uname</td>
</tr>
<tr>
<td>SYS_OPEN</td>
<td>max. number of files open on system</td>
</tr>
<tr>
<td>TMP_MAX</td>
<td>max. number of unique names generated by tmpnam</td>
</tr>
<tr>
<td>UID_MAX</td>
<td>max. value for a user-ID or group-ID</td>
</tr>
<tr>
<td>USI_MAX</td>
<td>max. decimal value of an unsigned</td>
</tr>
<tr>
<td>WORD_BIT</td>
<td>number of bits in a word or int</td>
</tr>
<tr>
<td>CHAR_MIN</td>
<td>min. integer value of a char</td>
</tr>
<tr>
<td>INT_MIN</td>
<td>min. decimal value of an int</td>
</tr>
<tr>
<td>LONG_MIN</td>
<td>min. decimal value of a long</td>
</tr>
<tr>
<td>SHRT_MIN</td>
<td>min. decimal value of a short</td>
</tr>
</tbody>
</table>

**parent-process-ID**

The parent-process-ID of a process is the process-ID of its creator. 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,
onoptionally followed by a file-name. A null string is undefined and may be con-
sidered 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
<dirname>::=<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. 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)].

process-group-leader

A process group leader is any process whose process-group-ID is the same as its
process-ID. Any process may detach itself from its current process-group and
become a new process-group-leader by calling the SETPGRP(BA_OS) routine. A
process inherits the process-group-ID of the process that created it [see
FORK(BA_OS) and EXEC(BA_OS)].

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.

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 5, the system service (BA_OS) routines or Chapter 6, the general library (BA_LIB) routines.

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

(BA_LIB) ctermid, fgetc, fgets, fprintf, fputc, fputs, fscanf, getchar, gets, getw, print, 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 getc and putc, 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 BUFSIZE specifies the size of the stdio buffers used by the particular implementation.

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 `TERMI0(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
console — system console interface

SYNOPSIS
/dev/console

DESCRIPTION
/dev/console is a generic name given to the system console. It is usually linked to a particular machine-dependent special file, and provides a basic I/O interface to the system console through the termio interface [see TERMIO(BA_ENV)].

SEE ALSO
TERMIO(BA_ENV).

LEVEL
Level 1.
NAME
null — the null file

SYNOPSIS
/dev/null

DESCRIPTION
Data written on a null special file are discarded.

Read operations from a null special file always return 0 bytes.

Output of a command is written to the special file /dev/null when the command is executed for its side effects and not for its output.

LEVEL
Level 1.
DEVTTY(BA_ENV)

NAME
tty — controlling terminal interface

SYNOPSIS
/dev/tty

DESCRIPTION
The file /dev/tty is, in each process, a synonym for the control-terminal associated with the process group of that process, if any. It is useful for programs that wish to be sure of writing messages on the terminal no matter how output has been redirected [see SYSTEM(BA_OS)]. It can also be used for programs that demand the name of a file for output when typed output is desired and as an alternative to identifying what terminal is currently in use.

APPLICATION USAGE
Normally, application programs should not need to use this file interface. The standard input, standard output and standard error files should be used instead. These file are accessed through the stdin, stdout and stderr stdio interfaces [see stdio-stream in Chapter 4 — Definitions].

SEE ALSO
TERMIO(BA_ENV).

LEVEL
Level 1.
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.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECHILD</td>
<td>No child processes. The <code>WAIT(BA_OS)</code> routine was executed by a process that had no existing or unwaited-for child processes.</td>
</tr>
<tr>
<td>EDEADLK</td>
<td>Deadlock avoided. The request would have caused a deadlock; the situation was detected and avoided.</td>
</tr>
<tr>
<td>EDOM</td>
<td>Math argument. The argument of a function in the math package is out of the domain of the function.</td>
</tr>
<tr>
<td>EEXIST</td>
<td>File exists. An existing file was mentioned in an inappropriate context (e.g., a call to the <code>LINK(BA_OS)</code> routine).</td>
</tr>
<tr>
<td>EFAULT</td>
<td>Bad address. The system encountered a hardware fault in attempting to use an argument of a routine. For example, <code>errno</code> potentially may be set to <code>EFAULT</code> 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.</td>
</tr>
<tr>
<td>EFBIG</td>
<td>File too large. The size of a file exceeded the maximum file size, <code>{FCHR_MAX}</code> (see <code>ULIMIT(BA_OS)</code>).</td>
</tr>
<tr>
<td>EINTR</td>
<td>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.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid argument. Some invalid argument (e.g., dismounting a non-mounted device; mentioning an undefined signal in a call to the <code>SIGNAL(BA_OS)</code> or <code>KILL(BA_OS)</code> routine). Also set by math routines.</td>
</tr>
<tr>
<td>EIO</td>
<td>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.</td>
</tr>
<tr>
<td>EISDIR</td>
<td>Is a directory. An attempt was made to write on a directory.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>Too many open files in a process. No process may have more than <code>{OPEN_MAX}</code> file descriptors open at a time.</td>
</tr>
</tbody>
</table>
**ERRNO(BA_ENV)**

**ENLINK** Too many links
An attempt to make more than the maximum number of links, \([\text{LINK\_MAX}]\), to a file.

**ENFILE** Too many open files in the system
The system file table is full (i.e., \([\text{SYS\_OPEN}]\) 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.

**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.
ERRNO(BA_ENV)

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.
NAME

envvar — environmental variables

DESCRIPTION

When a process begins execution, the EXEC(BA_OS) routines make available an array of strings called the environment [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. The following environmental variables can be used by applications and are expected to be set in the target run-time environment.

Variable Use

HOME Full path-name of the user's home-directory, the user's initial-working-directory [see PASSWD(BA_ENV)].

PATH Colon-separated ordered list of path-names that determine the search sequence used in locating files [see SYSTEM(BA_OS)].

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

TZ Time-zone information. TZ must be a three-letter, local time-zone abbreviation, followed by a number (an optional minus sign, for time-zones east of Greenwich, followed by a series of digits) that is the difference in hours between this time-zone and Greenwich Mean Time. This may be followed by an optional three-letter daylight local time-zone. For example, EST5EDT for Eastern Standard, Eastern Daylight Savings Time.

Other variables might be set in a particular environment but are not required to be included in the Base System.

SEE ALSO

EXEC(BA_OS), SYSTEM(BA_OS), FILSYS(BA_ENV).

FUTURE DIRECTIONS

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

LEVEL

Level 1.
NAME
file system — directory tree structure

DESCRIPTION

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

```
/   
|   |
bin dev etc tmp usr
```

The following guidelines apply to the contents of these directories:

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

- `/bin` holds executable system commands (utilities), if any.

- `/dev` holds special device files.

- `/etc` holds system data files, such as `/etc/passwd`.

- `/tmp` holds temporary files created by utilities in `/bin` and by other system processes.

- `/usr/bin` holds (user-level) executable application and system commands.

- `/usr/tmp` holds temporary files created by applications and the system.

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

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

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

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

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

The `/etc/profile` file may contain a string assignment of the `PATH` and `TZ` variables defined in `ENVVAR(BA_ENV)`.

**FUTURE DIRECTIONS**

The following directory structure and guidelines are proposed for applications ("add-ons") that are to be installed on a system:

```
/etc
    bin  etc  lib  opt  tmp
          x  y
```

- `/usr/etc` would hold data and log files for commands in `/usr/bin`.
- `/usr/lib` would hold any executable files for commands in `/usr/bin`.
- `/usr/opt` would hold sub-directories for each add-on to hold data files private to the add-on (e.g., add-on `x`)
- `/usr/opt/x` would hold files and/or directories private to add-on `x`, `/usr/opt/y` would hold files and/or directories private to add-on `y`.

**LEVEL**

Level 1.
NAME

passwd — password file

SYNOPSIS

/etc/passwd

DESCRIPTION

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

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

This ASCII file resides in directory /etc. It has general read permission and can be used, for example, to map numerical user-IDs to names.

Each field within each user's entry is separated from the next by a colon. Fields 2, 4, and 7 may be empty. However, if they are not empty, they must be used for their stated purpose. Field 5 is a free field that is implementation-specific. Fields beyond 7 are also free but may be standardized in the future. Each user's entry is separated from the next by a newline.

The name is a character string that identifies a user. Its composition should follow the same rules used for file-names.

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

SEE ALSO

CRYPT(3LIB).

LEVEL

Level 1.
NAME
termio — general terminal interface

SYNOPSIS
#include <termio.h>

ioctl(fildes, request, arg)
struct termio *arg;

ioctl(fildes, request, arg)
int arg;

DESCRIPTION
System V supports a general interface for asynchronous communications ports that is hardware-independent. The remainder of this section discusses the common features of this interface.

When a terminal file is opened, it normally causes the process to wait until a connection is established. Typically, these files are opened by the system initialization process and become the standard input, standard output, and standard error files [see stdio-stream in Chapter 4 — Definitions]. The very first terminal file opened by the process-group-leader but not already associated with a process-group becomes the control-terminal for that process-group. The control-terminal plays a special role in handling quit and interrupt signals [see below]. The control-terminal is inherited by a new process during a FORK(BA_OS) or EXEC(BA_OS) operation. A process can break this association by changing its process-group with the SETPGRP(BA_OS) routine.

A terminal associated with one of these files ordinarily operates in full-duplex mode. This means characters may be typed at any time, even while output is occurring. Characters are only lost when the system's character input buffers become completely full, or when an input line exceeds {MAX_CHAR}, the maximum allowable number of input characters. When the input limit is reached, all the saved characters may be thrown away without notice.

Normally, terminal input is processed in units of lines. A line is delimited by the new-line (ASCII LF) character, end-of-file (ASCII EOT) character, or end-of-line character. This means that a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters may be requested in a read, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

Some characters have special meaning when input. For example, during input, erase and kill processing is normally done. The ERASE character erases the last character typed, except that it will not erase beyond the beginning of the line. Typically, the default ERASE character is the character #. The KILL character kills (deletes) the entire input line, and optionally outputs a new-line character. Typically, the default KILL character is the character @. Both characters operate on a key-stroke basis independently of any backspacing or tabbing.
**Special Characters.**

Some characters have special functions on input. These functions and their typical default character values are summarized below:

**INTR** (Typically, rubout or ASCII DEL) generates an *interrupt* signal, which is sent to all processes with the associated control-terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location [see SIGNAL(BA_OS)].

**QUIT** (Typically, control-\ or ASCII FS) generates a *quit* signal. Its treatment is identical to the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but the abnormal termination routines will be executed.

**ERASE** (Typically, the character #) erases the preceding character. It will not erase beyond the start of a line, as delimited by an EOF, EOL or NL character.

**KILL** (Typically, the character @) deletes the entire line, as delimited by an EOF, EOL or NL character.

**EOF** (Typically, control-d or ASCII EOT) may be used to generate an EOF, from a terminal. When received, all the characters waiting to be read are immediately passed to the program, without waiting for a new-line, and the EOF is discarded. Thus, if there are no characters waiting, which is to say the EOF occurred at the beginning of a line, zero characters will be passed back, which is the standard end-of-file indication.

**NL** (ASCII LF) is the normal line delimiter. It can not be changed or escaped.

**EOL** (Typically, ASCII NUL) is an additional line delimiter, like NL. It is not normally used.

**STOP** (Typically, control-s or ASCII DC3) is used to temporarily suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.

**START** (Typically, control-q or ASCII DC1) is used to resume output suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The START/STOP characters can not be changed or escaped.
MIN Used to control terminal I/O during raw mode (ICANON off) processing [see the MIN/TIME Interaction section below].

TIME Used to control terminal I/O during raw mode (ICANON off) processing [see the MIN/TIME Interaction section below].

The ERASE, KILL, and EOF characters may be entered literally, and their special meaning escaped, by preceding them with the escape character `. In this case, no special function is performed. Also the escape character is not read as input.

When one or more characters are written, they are transmitted to the terminal as soon as previously-written characters have finished typing. Input characters are echoed by putting them in the output queue as they arrive. If a process produces characters more rapidly than they can be typed, it will be suspended when its output queue exceeds some limit. When the queue has drained down to some threshold, the program is resumed.

When a modem disconnect is detected, a hang-up signal, SIGHUP, is sent to all processes that have this terminal as the control-terminal. Unless other arrangements have been made, this signal causes the processes to terminate. If the hang-up signal is ignored, any subsequent read returns with an end-of-file indication. Thus, programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

IOCTL(BA_OS) Requests.
Several IOCTL(BA_OS) requests apply to terminal files and use the structure termio which is defined by the <termio.h> header file.

The primary IOCTL(BA_OS) requests to a terminal have the form:

```
ioctl(fildes, request, arg)
struct termio *arg;
```

The requests using this form are:

**TCGETA** Get the parameters associated with the terminal and store in the structure termio referenced by arg.

**TCSETA** Set the parameters associated with the terminal from the structure termio referenced by arg. The change is immediate.

**TCSETAW** Wait for the output to drain before setting the new parameters. This form should be used when changing parameters that will affect output.

**TCSETAF** Wait for the output to drain, then flush the input queue and set the new parameters.
Additional ioctl(BA_OS) requests to a terminal have the form:

```c
ioctl(fildes, request, arg)
int arg;
```

The requests using this form are:

**TCSBRK**  Wait for the output to drain.
If `arg` is 0, then send a break (zero bits for 0.25 seconds).

**TCXONC**  Start/stop control.
If `arg` is 0, suspend output; if 1, restart suspended output.

**TCFLSH**  Flush queues
If `arg` is 0, flush the input queue; if 1, flush the output queue;
if 2, flush both the input and output queues.

The structure `termio` includes the following members:

```c
unsigned short c_iflag;  /* input modes */
unsigned short c_oflag;  /* output modes */
unsigned short c_cflag;  /* control modes */
unsigned short c_lflag;  /* local modes */
char c_line;  /* line-discipline */
unsigned char c_cc[NCC];  /* control chars */
```

The special control-characters are defined by the array `c_cc`. The symbolic name `NCC` is the size of the control-character array and is also defined by the `<termio.h>` header file. The relative positions, subscript names and typical default values for each entry are as follows:

<table>
<thead>
<tr>
<th>Subscript</th>
<th>Character</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VINTR</td>
<td>ASCII DEL</td>
</tr>
<tr>
<td>1</td>
<td>VQUIT</td>
<td>ASCII FS</td>
</tr>
<tr>
<td>2</td>
<td>VERASE</td>
<td>#</td>
</tr>
<tr>
<td>3</td>
<td>VKILL</td>
<td>@</td>
</tr>
<tr>
<td>4</td>
<td>VEOF</td>
<td>ASCII EOT</td>
</tr>
<tr>
<td>5</td>
<td>VMIN</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>VEOL</td>
<td>ASCII NUL</td>
</tr>
<tr>
<td>5</td>
<td>VTIME</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>reserved</td>
<td></td>
</tr>
</tbody>
</table>

**Input Modes.**
The following values for the field `c_iflag` define the basic terminal input control:

**IGNBRK**  Ignore break condition.
If `IGNBRK` is set, the break condition (a character framing error with data all zeros) is ignored, that is, not put on the input queue and therefore not read by any process. Otherwise, see BRKINT.
**BRKINT** Signal interrupt on break.
If **BRKINT** is set, the break condition will generate an interrupt signal and flush both the input and output queues.

**IGNPAR** Ignore characters with parity errors.
If **IGNPAR** is set, characters with other framing and parity errors are ignored.

**PARMRK** Mark parity errors.
If **PARMRK** is set, a character with a framing or parity error which is not ignored is read as the three-character sequence: 0377, 0, X, where 0377, 0 is a two-character flag preceding each sequence and X is the data of the character received in error. To avoid ambiguity in this case, if **ISTRIP** is not set, a valid character of 0377 is read as 0377, 0377.
If **PARMRK** is not set, a framing or parity error which is not ignored is read as the character ASCII NUL (ASCII code 0).

**INPCK** Enable input parity check.
If **INPCK** is set, input parity checking is enabled.
If **INPCK** is not set, input parity checking is disabled allowing output parity generation without input parity errors.

**ISTRIP** Strip character.
If **ISTRIP** is set, valid input characters are first stripped to 7-bits, otherwise all 8-bits are processed.

**INLCR** Map NL to ASCII CR on input.
If **INLCR** is set, a received NL character is translated into a ASCII CR character.

**IGNCR** Ignore ASCII CR.
If **IGNCR** is set, a received ASCII CR character is ignored (not read).

**ICRNL** Map ASCII CR to NL on input.
If **ICRNL** is set, a received ASCII CR character is translated into a NL character.

**IUCLC** Map upper-case to lower-case on input.
If **IUCLC** is set, a received upper-case alphabetic character is translated into lower-case.

**IXON** Enable start/stop output control.
If **IXON** is set, start/stop output control is enabled. A received STOP character will suspend output and a received START character will restart output. All start/stop characters are ignored and not read.
TERMIO(BA_ENV)

**IXANY** Enable any character to restart output.
If IXANY is set, any input character, will restart output which has been suspended.

**IXOFF** Enable start/stop input control.
If IXOFF is set, the system will transmit START/STOP characters when the input queue is nearly empty/full.

The initial input control value is all bits clear.

**Output Modes.**
The following values for the field c_oflag define the system treatment of output:

**OPOST** Postprocess output.
If OPOST is set, output characters are post-processed as indicated by the remaining flags; otherwise characters are transmitted without change.

**OLCUC** Map lower case to upper on output.
If OLCUC is set, a lower-case alphabetic character is transmitted as the corresponding upper-case character. This function is often used in conjunction with IUCLC.

**ONLCR** Map NL to ASCII CR-NL on output.
If ONLCR is set, the NL character is transmitted as the ASCII CR-NL character pair.

**OCRNL** Map ASCII CR to NL on output.
If OCRNL is set, the ASCII CR character is transmitted as the NL character.

**ONOCR** No ASCII CR output at column 0.
If ONOCR is set, no ASCII CR character is transmitted when at column 0 (first position).

**ONLRET** NL performs ASCII CR function.
If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer will be set to 0 and the delays specified for ASCII CR will be used. Otherwise the NL character is assumed to do just the line-feed function; the column pointer will remain unchanged. The column pointer is also set to 0 if the ASCII CR character is actually transmitted.

**OFILL** Use fill-characters for delay.
If OFILL is set, fill-characters will be transmitted for delay instead of a timed delay. This is useful for high baud-rate terminals that need only a minimal delay.

**OFDEL** Fill is ASCII DEL, else ASCII NUL.
If OFDEL is set, the fill-character is ASCII DEL, otherwise ASCII NUL.
The delay-bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay.

The actual delays depend on line-speed and system-load.

**NLDLY** New-line delay lasts about 0.10 seconds.
If ONLRET is set, the carriage-return delays are used instead of the new-line delays.
If OFILL is set, two fill-characters will be transmitted.

Select new-line delays:
- **NL0** New-Line character type 0
- **NL1** New-Line character type 1

**CRDLY** Carriage-return delay type 1 is dependent on the current column position, type 2 is about 0.10 seconds, and type 3 is about 0.15 seconds.
If OFILL is set, delay type 1 transmits two fill-characters, and type 2, four fill-characters.

Select carriage-return delays:
- **CR0** Carriage-return delay type 0
- **CR1** Carriage-return delay type 1
- **CR2** Carriage-return delay type 2
- **CR3** Carriage-return delay type 3

**TABDLY** Horizontal-tab delay type 1 is dependent on the current column position, type 2 is about 0.10 seconds, and type 3 specifies that tabs are to be expanded into spaces.
If OFILL is set, two fill-characters will be transmitted for any delay.

Select horizontal-tab delays:
- **TAB0** Horizontal-tab delay type 0
- **TAB1** Horizontal-tab delay type 1
- **TAB2** Horizontal-tab delay type 2
- **TAB3** Expand tabs to spaces

**BSDLY** Backspace delay lasts about 0.05 seconds.
If OFILL is set, one fill-character will be transmitted.

Select backspace delays:
- **BS0** Backspace delay type 0
- **BS1** Backspace delay type 1

**VTDLY** Vertical-tab delay lasts about 2.0 seconds.

Select vertical-tab delays:
- **VT0** Vertical-tab delay type 0
- **VT1** Vertical-tab delay type 1
TERMIO(BA_ENV)

**FFDLY**  
Form-feed delay lasts about 2.0 seconds.

Select form-feed delays:
- **FF0**  Form-feed delay type 0
- **FF1**  Form-feed delay type 1

The initial output control value is all bits clear.

**Control Modes.**  
The following values for the field c_flag define the hardware control of the terminal:

- **CBAUD**  Specify the baud-rate.  
The zero baud-rate, **B0**, is used to hang up the connection. If **B0** is specified, the data-terminal-ready signal will not be asserted. Normally, this will disconnect the line. For any particular hardware, unsupported speed changes are ignored.

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

- **CSIZE**  Specify the character size in bits for both transmission and reception. This size does not include the parity-bit, if any.

Select character size:
- **CS5**  5-bits
- **CS6**  6-bits
- **CS7**  7-bits
- **CS8**  8-bits

- **CSTOPB**  Send two stop-bits, else one.  
If **CSTOPB** is set, two stop-bits are used, otherwise one stop-bit. For example, at 110 baud, two stop-bits are normally used.

- **CREASE**  Enable receiver.  
If **CREASE** is set, the receiver is enabled. Otherwise no characters will be received.
TERMIO(BA_ENV)

**PARENB**  Enable parity.
If **PARENB** is set, parity generation and detection is enabled and a parity-bit is added to each character.

**PARODD**  Specify odd parity, else even.
If parity is enabled, the **PARODD** flag specifies odd parity if set, otherwise even parity is used.

**HUPCL**  Hang up on last close.
If **HUPCL** is set, the modem control lines for the port will be lowered when the last process with the line open closes it or terminates. That is, the data-terminal-ready signal will not be asserted.

**CLOCAL**  Local line, else dial-up.
If **CLOCAL** is set, the line is assumed to be a local, direct connection with no modem control. Otherwise modem control is assumed.

Under normal circumstances, an **OPEN(BA_OS)** operation will wait for the modem connection to complete. However, if the **O_NDELAY** flag is set, or **CLOCAL** is set, the **OPEN(BA_OS)** operation will return immediately without waiting for the connection. For those files on which the connection has not been established, or has been lost, and for which **CLOCAL** is not set, both **READ(BA_OS)** and **WRITE(BA_OS)** operations will return a zero character count. For the **READ(BA_OS)** operation, this is equivalent to an end-of-file condition. The initial hardware control value after the **OPEN(BA_OS)** operation is implementation-dependent.

**Local Modes and Line Discipline.**

The field **c_iflag** of the structure **termio** is used by the line-discipline to control terminal functions. The basic line-discipline, **c_line** set to 0, provides the following:

**ISIG**  Enable signals.
If **ISIG** is set, each input character is checked against the special control characters **INTR** and **QUIT**. If an input character matches one of these control characters, the function associated with that character is performed. If **ISIG** is not set, no checking is done. Thus these special input functions are possible only if **ISIG** is set. These functions may be disabled individually by changing the value of the control character to an unlikely or impossible value (e.g., 0377).

**ICANON**  Canonical input (**ERASE** and **KILL** processing).
If **ICANON** is set, canonical processing is enabled. This enables the **ERASE** and **KILL** edit functions, and the assembly of input characters into lines delimited by the **EOF**, **EOL** or **NL** characters. If **ICANON** is not set, read requests are satisfied directly from...
the input queue. A read will not be satisfied until at least MIN characters have been received or the time-out value TIME has expired between characters [see the MIN/TIME Interaction section below]. This allows fast bursts of input to be read efficiently while still allowing single character input. The MIN and TIME values are stored in the position for the EOF and EOL characters, respectively. The time-value is expressed in units of 0.10 seconds.

**XCASE** Canonical upper/lower presentation.

If both **XCASE** and **ICANON** are set, an upper-case letter is input by preceding it with the character \, and is output preceded by the character \. In this mode, the following escape sequences are generated on output and accepted on input:

```
for:          use:
\             \\n|             \|
\~             \^
\{             \(
\}             \)
\             \
```

For example, A is input as \a, \n as \\n, and \N as \\
.

**ECHO** Enable echo.

If **ECHO** is set, characters are echoed back to the terminal as received.

When **ICANON** is set, the following echo functions are possible:

**ECHOE** Echo the ERASE character as ASCII BS-SP-BS.

If both **ECHOE** and **ECHO** are set, the ERASE character is echoed as ASCII BS-SP-BS, which will clear the last character from a CRT screen.

If **ECHOE** is set but **ECHO** is not set, the ERASE character is echoed as ASCII SP-BS.

**ECHOK** Echo the NL character after the KILL character.

If **ECHOK** is set, the NL character will be echoed after the KILL character to emphasize that the line will be deleted. Note that an escape character preceding the ERASE character or the KILL character removes any special function.

**ECHONL** Echo the NL character.

If **ECHONL** is set, the NL character will be echoed even if **ECHO** is not set. This is useful for terminals set to local-echo (also called half-duplex). Unless escaped, the EOF character is not echoed. Because ASCII EOT is the default EOF character, this prevents terminals that respond to ASCII EOT from hanging up.
NOFLSH Disable flush after interrupt or quit.
  If NOFLSH is set, the normal flush of the input and output queues associated with the quit and interrupt characters will not be done.

The initial line-discipline control value is all bits clear.

**MIN/TIME Interaction.**
MIN represents the minimum number of characters that should be received when the read is satisfied (i.e., the characters are returned to the user). TIME is a timer of 0.10 second granularity used to time-out bursty and short-term data transmissions. The four possible values for MIN and TIME and their interactions follow:

1. **MIN>0, TIME>0.** In this case, TIME serves as an inter-character timer activated after the first character is received, and reset upon receipt of each character. MIN and TIME interact as follows:
   - As soon as one character is received the inter-character timer is started.
   - If MIN characters are received before the inter-character timer expires the read is satisfied.
   - If the timer expires before MIN characters are received the characters received to that point are returned to the user.
   
   A READ(BA_OS) operation will sleep until the MIN and TIME mechanisms are activated by the receipt of the first character; thus, at least one character must be returned.

2. **MIN>0, TIME=0.** In this case, because TIME=0, the timer plays no role and only MIN is significant. A READ(BA_OS) operation is not satisfied until MIN characters are received.

3. **MIN=0, TIME>0.** In this case, because MIN=0, TIME no longer serves as an inter-character timer, but now serves as a read timer that is activated as soon as the READ(BA_OS) operation is processed (in canon). A READ(BA_OS) operation is satisfied as soon as a single character is received or the timer expires, in which case, the READ(BA_OS) operation will not return any characters.

4. **MIN=0, TIME=0.** In this case, return is immediate. If characters are present, they will be returned to the user.

SEE ALSO
FORK(BA_OS), IOCTL(BA_OS), SETPGRP(BA_OS), SIGNAL(BA_OS).

LEVEL
Level 1.
Chapter 6
OS Service Routines
ABORT(BA_OS)

NAME
abort — generate an abnormal process termination

SYNOPSIS
int abort()

DESCRIPTION
The function abort first closes all open files if possible, then causes a signal 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 should not be caught or ignored by applications.

SEE ALSO
EXIT(BA_OS), SIGNAL(BA_OS).

FUTURE DIRECTIONS
The function abort will send the SIGABRT signal rather than the SIGIOT signal.

LEVEL
Level 1.
NAME
access — determine accessibility of a file

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

DESCRIPTION
The function access checks the named file for accessibility according to the bit-pattern contained in amode, using 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 bit-pattern contained in amode is constructed as follows:

  04  read
  02  write
  01  execute (search)
  00  check existence of file

Thus, the argument amode should be the sum of the values of the access modes to be checked.

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.
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.
ACCESS(BA_OS)

SEE ALSO
CHMOD(BA_OS), STAT(BA_OS).

FUTURE DIRECTIONS

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

The <unistd.h> header file will define the following symbolic constants for the argument amode to the function access:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_OK</td>
<td>test for read permission.</td>
</tr>
<tr>
<td>W_OK</td>
<td>test for write permission.</td>
</tr>
<tr>
<td>X_OK</td>
<td>test for execute permission.</td>
</tr>
<tr>
<td>F_OK</td>
<td>test for existence of file.</td>
</tr>
</tbody>
</table>

LEVEL

Level 1.
NAME
alarm — set a process alarm clock

SYNOPSIS
unsigned alarm(sec)
unsigned sec;

DESCRIPTION
The function alarm instructs the alarm clock of the calling-process to send
the signal SIGALRM to the calling-process after the number of real time
seconds specified by sec have elapsed [see SIGNAL(BA_OS)].

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

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

The FORK(BA_OS) routine sets the alarm clock of a new process to 0. A
process created by the EXEC(BA_OS) family of routines inherits the time left
on the old process's alarm clock.

RETURN VALUE
If successful, the function alarm will return the amount of time previously
remaining in the alarm clock of the calling-process.

SEE ALSO
EXEC(BA_OS), FORK(BA_OS), PAUSE(BA_OS), SIGNAL(BA_OS).

LEVEL
Level 1.
NAME
chdir — change working directory

SYNOPSIS
int chdir(path)
char *path;

DESCRIPTION
The function chdir causes the named directory to become the current
working directory and the starting point for path-searches for path-names not
beginning with /.

The argument path points to the path-name of a directory.

RETURN VALUE
If successful, the function chdir will return 0; otherwise, it will return
−1, the current-working-directory will be unchanged and errno will indi­
cate the error.

ERRORS
Under the following conditions, the function chdir will fail and will set
errno to:
ENOTDIR if a component of the path-name is not a directory.
ENOENT if the named directory does not exist.
EACCES if any component of the path-name denies search permission.

LEVEL
Level 1.
NAME
chmod — change mode of file

SYNOPSIS
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.

Access permission bits are interpreted as follows; the value of the argument mode should be the sum of the values of the desired permissions:

- 04000 Set user-ID on execution.
- 02000 Set group-ID on execution.
- 01000 Reserved.
- 00400 Read by owner.
- 00200 Write by owner.
- 00100 Execute (search if a directory) by owner.
- 00040 Read by group.
- 00020 Write by group.
- 00010 Execute (search) by group.
- 00004 Read by others (i.e., anyone else).
- 00002 Write by others.
- 00001 Execute (search) by others.

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, mode bit 02000 (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.

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.
CHMOD(BA_OS)

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.
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).

FUTURE DIRECTIONS
Symbolic constants defining the access permission bits will be added to the <sys/stat.h> header file and should be used to construct mode.

Enforcement-mode file and record-locking will be added:

If the mode bit 02000 (set group-ID on execution) is set and the mode bit 01000 (execute or search by group) is not set, enforcement-mode file and record-locking will exist on an ordinary-file. This may affect future calls to OPEN(BA_OS), CREAT(BA_OS), READ(BA_OS) and WRITE(BA_OS) routines on this file.

LEVEL
Level 1.
NAME
chown — change owner and group of a file

SYNOPSIS
int chown(path, owner, group)
char *path;
int owner, group;

DESCRIPTION
The function \texttt{chown} sets the owner-ID and group-ID of the named file to
the numeric values contained in \texttt{owner} and \texttt{group}, respectively.
The argument \texttt{path} points to a path-name naming a file.
Only processes with effective-user-ID equal to the file-owner or super-user
may change the ownership of a file.
If the function \texttt{chown} is invoked successfully by other than the super-user,
the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000
respectively, will be cleared. (This prevents ordinary users from making
themselves effectively other users or members of a group to which they don't
belong.)

RETURN VALUE
If successful, the function \texttt{chown} will return 0; otherwise, it will return
-1, the owner and group of the named file will remain unchanged and
\texttt{errno} will indicate the error.

ERRORS
Under the following conditions, the function \texttt{chown} will fail and will set
\texttt{errno} to:
\texttt{ENOTDIR} if a component of the path-prefix is not a directory.
\texttt{ENOENT} if the named file does not exist.
\texttt{EACCES} if a component of the path-prefix denies search permission.
\texttt{EPERM} if the effective-user-ID does not match the owner of the file and
the effective-user-ID is not super-user.
\texttt{EROFS} if the named file resides on a read-only file system.

SEE ALSO
CHMOD(\texttt{BA\_OS}).

LEVEL
Level 1.
CLOSE(BA_OS)

NAME
close — close a file-descriptor

SYNOPSIS
int close(fildes)
int fildes;

DESCRIPTION
The function close closes the file-descriptor indicated by fildes.
The argument fildes is an open file-descriptor [see file-descriptor in Chapter 4 — Definitions].
All outstanding record-locks on the file indicated by fildes that are owned by the calling-process are removed.

RETURN VALUE
If successful, the function close will return 0; otherwise, it will return -1 and errno will indicate the error.

ERRORS
Under the following conditions, the function close will fail and will set errno to:

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

APPLICATION USAGE
Normally, applications should use the stdio routines to open, close, read and write files. Thus, an application that had used the OPEN(BA_OS) stdio routine to open a file would use the corresponding FCLOSE(BA_OS) stdio routine rather than the CLOSE(BA_OS) routine.

The record and file locking features are an update that followed System V Release 1.0 and System V Release 2.0.

SEE ALSO
CREAT(BA_OS), DUP(BA_OS), EXEC(BA_OS), FCNTL(BA_OS), OPEN(BA_OS), PIPE(BA_OS).

LEVEL
Level 1.
NAME
creat — create a new file or rewrite an existing one

SYNOPSIS
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.

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.
CREAT(BA_OS)

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.
EACCES if a component of the path-prefix denies search permission, or if the file does not exist and the directory in which the file is to be created does not permit writing, or 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.

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).

FUTURE DIRECTIONS
Symbolic constants defining the access permission bits will be defined by the <sys/stat.h> header file and should be used to construct mode.

Enforcement-mode file and record locking features will be added:

The function creat will set errno to EAGAIN if the file exists, enforcement-mode file and record-locking is set and there are outstanding record-locks on the file [see CHMOD(BA_OS)].

LEVEL
Level 1.
NAME

dup — duplicate an open file-descriptor

SYNOPSIS

int dup(fildes)
int fildes;

DESCRIPTION

The function dup returns a new file-descriptor having the following in common with the original:

Same open file (or pipe).
Same file-pointer (i.e., both file-descriptors share one file-pointer).
Same access mode (read, write or read/write).

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

The new file-descriptor is set to remain open across calls to the EXEC(BA_OS) routines [see FCNTL(BA_OS)].

The file-descriptor returned is the lowest one available.

RETURN VALUE

If successful, the function dup 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 dup will fail and will set errno to:

EBADF if fildes is not a valid open file-descriptor.
EMFILE if (OPEN_MAX) file-descriptors are currently open in the calling-process.

SEE ALSO

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

LEVEL

Level 1.
EXEC(BA_OS)

NAME
execl, execv, 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 execl(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 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 `exec` and `execv`, a pointer to the environment of the calling-process is made available in the global cell:

```c
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 caught by the calling-process will be set to the default action in the new process [see `SIGNAL(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)` and `SIGNAL(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)`]
- (file-locks [see `FCNTL(BA_OS)` and `LOCKF(BA_OS)`])
EXEC(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.
- **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.
- **ENOMEM** 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.

**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), TIMES(BA_OS), ULIMIT(BA_OS), UMASK(BA_OS).

**LEVEL**

Level 1.

Page 72 Base System Definition
EXEC(BA_OS)

NAME
exit, _exit — terminate process

SYNOPSIS
void exit(status)
int status;

void _exit(status)
int status;

DESCRIPTION
The function exit may cause cleanup actions before the process exits [see
FCLOSE(BA_OS)]. The function _exit does not. The functions exit and
_exit terminate the calling-process with the following consequences:

All of the file-descriptors open in the calling-process are closed.

If the parent-process of the calling-process is executing a WAIT(BA_OS)
routine, it is notified of the calling-process's termination and the low-order
eight bits (i.e., bits 0377) of status are made available to it. If the
parent is not waiting, the child's status will be made available to it when
the parent subsequently executes the WAIT(BA_OS) routine.

If the parent-process of the calling-process is not executing a WAIT(BA_OS)
routine, the calling-process is transformed into a zombie-process. A
zombie-process is an inactive process that has no process space allocated
to it, and it will be deleted at some later time when its parent executes
the WAIT(BA_OS) routine.

Terminating a process by exiting does not terminate its children. The
parent-process-ID of all of the calling-process's existing child-processes
and zombie-processes is set to the process-ID of a special system-process.
That is, these processes are inherited by a special system-process.

If the calling-process is a process-group-leader, and is associated with a
controlling-terminal [see TERMIO(BA_ENV)], the SIGHUP signal is sent to
each process that has a process-group-ID and tty-group-ID equal to that
of the calling-process.

RETURN VALUE
Neither the function exit nor the function _exit will return a value.

APPLICATION USAGE
Normally applications should use exit rather than _exit.

SEE ALSO
SIGNAL(BA_OS), WAIT(BA_OS).

LEVEL
Level 1.
NAME
fclose, fflush — close or flush a stream

SYNOPSIS
#include <stdio.h>
int fclose(stream)
FILE *stream;

int fflush(stream)
FILE *stream;

DESCRIPTION
The function *fclose* causes any buffered data for the named *stream* to be written out, and the *stream* to be closed.

The function *fclose* is performed automatically for all open files upon calling the EXIT(BA_OS) routine.

The function *fflush* causes any buffered data for the named *stream* to be written to that file. The *stream* remains open.

RETURN VALUE
The functions *fclose* and *fflush* will return 0 for success, and EOF if any error (such as trying to write to a file that has not been opened for writing) was detected.

SEE ALSO
CLOSE(BA_OS), EXIT(BA_OS), FOPEN(BA_OS), SETBUF(BA_LIB).

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
[see OPEN(BA_OS)].

F_SETFL Set file status flags to arg. Only the flags O_NDELAY and O_APPEND may be set with fcntl.
The following commands are used for file-locking and 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 (1_type), starting offset (1_whence), relative offset (1_start), size (1_len), and process-ID (1_pid):

```c
short 1_type; /* F_RDLCK, F_WRLCK, F_UNLCK */
short 1_whence; /* flag for starting offset */
long 1_start; /* flag for starting offset in bytes */
long 1_len; /* if 0 then until EOF */
short 1_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 \( _\text{whence} \) is 0, 1 or 2 to indicate that the relative offset, \( _\text{start} \) bytes, will be measured from the start of the file, current position or end of the file, respectively. The value of \( _\text{len} \) is the number of consecutive bytes to be locked. The process-ID \( _\text{pid} \) field is only used with \texttt{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 \( _\text{len} \) to zero (0). If such a lock also has \( _\text{start} \) 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 \texttt{FORK(BA_OS)} routine.

**RETURN VALUE**
If successful, the function \texttt{fcntl} will return a value that depends on \texttt{cmd} as follows:

\begin{itemize}
  \item \texttt{F_DUPFD} a new file-descriptor.
  \item \texttt{F_GETFD} a value of flag (only the low-order bit is defined).
  \item \texttt{F_SETFD} a value other than \(-1\).
  \item \texttt{F_GETFL} a value of file flags.
  \item \texttt{F_SETFL} a value other than \(-1\).
  \item \texttt{F_GETLK} a value other than \(-1\).
  \item \texttt{F_SETLK} a value other than \(-1\).
  \item \texttt{F_SETLKW} a value other than \(-1\).
\end{itemize}

If unsuccessful, the function \texttt{fcntl} will return \(-1\) and \texttt{errno} will indicate the error.
ERRORS

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

- **EBADF** if `fd` is not a valid open file-descriptor.
- **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 `arg` or the data it 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 there are no more file-locks available (too many segments are locked).
- **EDEADLK** if `cmd` is `F_SETLKW`, the lock is blocked by some lock from another process and putting the calling-process to sleep, waiting for that lock to become free, would cause a deadlock.

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:

```c
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 file and 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).

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

Enforcement-mode file-locking and record locking will be added:

If enforcement-mode file and record-locking is set and there are outstanding record-locks on the file, this may affect future calls to READ(BA_OS) and WRITE(BA_OS) routines on the file [see CHMOD(BA_OS)].

LEVEL
Level 1.
FERROR(BA_OS)

NAME
ferror, feof, clearerr, fileno — stream status inquiries

SYNOPSIS
#include <stdio.h>

int ferror(stream)
FILE *stream;

int feof(stream)
FILE *stream;

void clearerr(stream)
FILE *stream;

int fileno(stream)
FILE *stream;

DESCRIPTION
The function ferror determines if an I/O error has occurred when reading from or writing to the named stream.

The function feof determines if EOF has been detected when reading the named stream.

The function clearerr resets both the error and EOF indicator to false on the named stream. The EOF indicator is reset when the file pointer associated with stream is repositioned, e.g., by the FSEEK(BA_OS) orREWIND(BA_OS) routines, or can be reset with clearerr.

The function fileno gets the integer file-descriptor associated with the named stream [see OPEN(BA_OS)].

RETURN VALUE
The function ferror will return non-zero when an I/O error has previously occurred reading from or writing to the named stream; otherwise, the function ferror will return zero.

The function feof will return non-zero when EOF has previously been detected reading the named input stream; otherwise, the function feof will return zero.

The function fileno will return the integer file-descriptor number associated with the named stream.

APPLICATION USAGE
All of these functions are implemented as macros; they cannot be declared or redeclared.

The function fileno returns a file-descriptor that can be used with non-stdio routines, such as WRITE(BA_OS) and LSEEK(BA_OS) routines, to manipulate the associated file, but these routines are not recommended for use by application-programs.
SEE ALSO
OPEN(BA_OS), FOPEN(BA_OS).

LEVEL
Level 1.
FOPEN(BA_OS)

NAME
fopen, freopen, fdopen — open a stream

SYNOPSIS
#include <stdio.h>
FILE *fopen(path, type)
char *path, *type;
FILE *freopen(path, type, stream)
char *path, *type;
FILE *stream;
FILE *fdopen(fildes, type)
int fildes;
char *type;

DESCRIPTION
The function fopen opens the file named by path and associates a
stream with it [see stream in Chapter 4 — Definitions]. The function
fopen returns a pointer to the FILE structure associated with the stream.

The function freopen substitutes the named file in place of the open
stream. The original stream is closed, regardless of whether the open
ultimately succeeds. The function freopen returns a pointer to the
FILE structure associated with stream.

The function freopen is typically used to attach the preopened streams
associated with stdin, stdout and stderr to other files. The stan-
dard error output stream stderr is by default unbuffered but use of the
function freopen will cause it to become buffered or line-buffered.

The argument path points to a character-string that names the file to be
opened.

The argument type is a character-string having one of the following
values:

- r open for reading.
- w truncate or create for writing.
- a append; open for writing at the end of the file, or create for writ-
ing.
- r+ open for update (reading and writing).
- w+ truncate or create for update.
- a+ append; open or create for update (appending) to the end of the
  file.
When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening call to the FSEEK(BA_OS) or REWIND(BA_OS) routine, and input may not be directly followed by output without an intervening call to the FSEEK(BA_OS) or REWIND(BA_OS) routine or an input operation which encounters end-of-file.

When a file is opened for append (i.e., when type is a or a+) it is impossible to overwrite information already in the file. The FSEEK(BA_OS) routine may be used to reposition the file-pointer to any position in the file, but when output is written to the file, the current file-pointer is disregarded. All output is written at the end of the file. For example, if two separate processes open the same file for append, each process may write to the file without overwriting output being written by the other, and the output from the two processes would be interleaved in the file.

The function fdopen associates a stream with a file-descriptor, fildes. The type of stream given to fdopen must agree with the mode of the already open file. File-descriptors are obtained from the routines which open files but do not return pointers to a FILE structure stream. Streams are necessary input for many of the stdio routines.

RETURN VALUE
The functions fopen and freopen return a NULL pointer if path cannot be accessed or if type is invalid or if the file cannot be opened.

The function fdopen will return a NULL pointer if fildes is not an open file-descriptor or if type is invalid or if the file cannot be opened.

The function fopen or the function fdopen may also fail if there are no free stdio streams.

ERRORS
When the file cannot be opened, the function fopen or the function freopen will fail and will set errno to:

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

ENOENT if the named file does not exist or a component of the path-name should exist but does not.

EACCES if a component of the path-prefix denies search permission or type permission is denied for the named file.

EISDIR if the named file is a directory and type is write or read/write.

EROFS if the named file resides on a read-only file system and type is write or read/write.

ETXTBSY if the file is a pure procedure (shared text) file that is being executed and type is write or read/write.

EINTR if a signal was caught during the open operation.
FOPEN(BA_OS)

SEE ALSO
    CREAT(BA_OS), DUP(BA_OS), OPEN(BA_OS), PIPE(BA_OS), FCLOSE(BA_OS),
    FSEEK(BA_OS).

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:

- environment
- close-on-exec flag [see EXEC(BA_OS)]
- signal-handling settings (i.e., SIG_DFL, SIG_IGN, address)
- set-user-ID mode bit
- set-group-ID mode bit
- process-group-ID
- tty-group-ID [see EXIT(BA_OS) and SIGNAL(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's file-descriptors shares a common file-pointer with the corresponding file-descriptor of the parent-process.
- The child-process's utime, stime, cutime, and cstime are set to 0. The time left until an alarm clock signal is reset to 0.

(File-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.
FORK(BA_OS)

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), TIMES(BA_OS), ULIMIT(BA_OS), UMASK(BA_OS), WAIT(BA_OS).

LEVEL
Level 1.
NAME
fread, fwrite — buffered input/output

SYNOPSIS
#include <stdio.h>

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

int fwrite(ptr, size, nitems, stream)
char *ptr;
int size, 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 argument size is typically sizeof(*ptr), where the C operator sizeof gives the length of an item pointed to by ptr. If ptr points to a data type other than char it should be cast into a pointer to char.

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

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),

FUTURE DIRECTIONS
The type of the argument size to the functions fread and fwrite
will be declared through the typedef facility in a header file as
size_t.

LEVEL
Level 1.
NAME
fseek, rewind, ftell — reposition a file-pointer in a stream

SYNOPSIS
#include <stdio.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 as whence has the value 0, 1, or 2.

The call rewind(stream) is equivalent to the following:

fseek(stream, 0L, 0)

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

FUTURE DIRECTIONS
Symbolic constants for the values of whence will be defined by the <unistd.h> header file [see LSEEK(BA_OS)].

LEVEL
Level 1.
GETCWD(BA_OS)

NAME
getcwd — get path-name of current working directory

SYNOPSIS
char *getcwd(buf, size)
char *buf;
int size;

DESCRIPTION
The function getcwd returns a pointer to the current directory path-name. The value of size must be at least two greater than the length of the path-name to be returned.

RETURN VALUE
If size is not large enough or if an error occurs in a lower-level function, the function getcwd will return NULL and errno will indicate the error.

ERRORS
Under the following conditions, the function getcwd will fail and will set errno to:

EINVAL if size is zero
ERANGE if size not large enough to hold the path-name.

LEVEL
Level 1.
NAME
getpid, getpgrp, getppid — get process-ID, process-group-ID, and parent-process-ID

SYNOPSIS
int getpid()
int getpgrp()
int getppid()

DESCRIPTION
The function getpid returns the process-ID of the calling-process.
The function getpgrp returns the process-group-ID of the calling-process.
The function getppid returns the parent-process-ID of the calling-process.

SEE ALSO
EXEC(6), FORK(6), SETPGRP(6), SIGNAL(6).

LEVEL
Level 1.
NAME
getuid, geteuid, getgid, getegid — get real-user-ID, effective-user-ID, real-group-ID, and effective-group-IDs

SYNOPSIS
unsigned short getuid()
unsigned short geteuid()
unsigned short getgid()
unsigned short getegid()

DESCRIPTION
The function getuid returns the real-user-ID of the calling-process.
The function geteuid returns the effective-user-ID of the calling-process.
The function getgid returns the real-group-ID of the calling-process.
The function getegid returns the effective-group-ID of the calling-process.

SEE ALSO
SETUID(BA_OS).

LEVEL
Level 1.
NAME
ioctl — control device

SYNOPSIS
int ioctl(fildes, request, arg)
int fildes, request;

DESCRIPTION
The function ioctl performs a variety of control functions on devices. This call passes the request to a device-driver to perform device-specific control functions.

NOTE: This control is not frequently used and the basic input/output operations are performed by the READ(BA_OS) and WRITE(BA_OS) routines.

The argument fildes is an open file-descriptor that refers to a device.

The argument request selects the control function to be performed and will depend on the device being addressed.

The argument arg represents additional information that is needed by this specific device to perform the requested function. The data type of arg depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

In addition to device-specific functions, there are generic functions that are provided by more than one device-driver, for example, the general terminal interface [see TERMIO(BA_ENV)].

RETURN VALUE
If successful, the function ioctl will return a value that depends upon the device control function, but must be an integer value; otherwise, it will return -1 and errno will indicate the error.

ERRORS
Under the following conditions, the function ioctl will fail and will set errno to:

EBADF if fildes is not a valid open file-descriptor.
ENOTTY if fildes is not associated with a device-driver that accepts control functions.
EINTR if a signal was caught during the ioctl operation.

The function ioctl will also fail if the device-driver detects an error. In this case, the error is passed through ioctl without change to the caller. A particular device-driver might not have all of the following error cases.
IOCTL(BA_OS)

Under the following conditions, requests to standard device-drivers may fail and \texttt{errno} will be set to:

\begin{itemize}
  \item \texttt{EINVAL} if \texttt{request} or \texttt{arg} are not valid for this device.
  \item \texttt{EIO} if some physical I/O error has occurred.
  \item \texttt{ENXIO} if \texttt{request} and \texttt{arg} are valid for this device-driver, but the service requested can not be performed on this particular sub-device.
\end{itemize}

\textbf{SEE ALSO}

The specific device reference documents and generic devices such as the general terminal interface [see \texttt{TERMIO(BA_ENV)}].

\textbf{LEVEL}

Level 1.
NAME

kill — send a signal to a process or a group of processes

SYNOPSIS

#include <signal.h>
int kill(pid, sig)
int pid, sig;

DESCRIPTION

The function kill sends a signal to a process or a group of processes.

The signal that is to be sent is specified by the argument sig and is either one from the list given in SIGNAL(BA_OS), or 0. If sig is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of pid.

The process or group of processes to which the signal is to be sent is specified by the argument pid. The argument pid specifies the processes to receive the signal as follows:

- If pid is greater than 0, sig will be sent to the process whose process-ID is equal to pid.
- If pid is 0, sig will be sent to all processes, excluding special system processes, whose process-group-ID is equal to the process-group-ID of the sender.
- If pid is negative but not -1, sig will be sent to all processes whose process-group-ID is equal to the absolute value of pid.
- If pid is -1, sig will be sent to all processes, excluding special system-processes.

Of the processes specified by pid, only those where the real-user-ID or effective-user-ID of the sending-process matches the real-user-ID or effective-user-ID of the receiving-process will be sent the signal, unless the effective-user-ID of the sending-process is super-user.

RETURN VALUE

If successful, the function kill will return 0; otherwise, it will return -1, no signal will be sent and errno will indicate the error.

ERRORS

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

- EINVAL if sig is not a valid signal number or if sig is SIGKILL and pid is a special system-process.
- ESRCH if no process corresponding to pid can be found.
- EPERM if the user-ID of the sending-process is not super-user, and its real-user-ID (or effective-user-ID) does not match either the real-user-ID or effective-user-ID of the receiving-process.
KILL(BA_OS)

SEE ALSO
GETPID(BA_OS), SETGRP(BA_OS), SIGNAL(BA_OS).

FUTURE DIRECTIONS
EPERM will be returned in errno if sig is SIGKILL and pid is a special system-process.

LEVEL
Level 1.
NAME
   link — link to a file

SYNOPSIS
   int link(path1, path2)
   char *path1, *path2;

DESCRIPTION
   The function link creates a new link (directory entry) for the existing file.
   The argument path1 points to a path-name naming an existing file.
   The argument path2 points to a path-name naming the new directory entry to be created.

RETURN VALUE
   If successful, the function link will return 0; otherwise, it will return -1, no link will be created and errno will indicate the error.

ERRORS
   Under the following conditions, the function link will fail and will set errno to:

   ENOTDIR if a component of either path-prefix is not a directory.
   ENOENT  if a component of either path-name should exist but does not.
   EACCES  if a component of either path-prefix denies search permission, or if the requested link requires writing in a directory with a mode that denies write permission.
   EEXIST  if the link named by path2 exists.
   EPERM   if the file named by path1 is a directory and the effective-user-ID is not super-user.
   EXDEV   if the link named by path2 and the file named by path1 are on different logical devices (file-systems) and the implementation does not permit cross-device links.
   EROFS   if the requested link requires writing in a directory on a read-only file-system.
   EMLINK  if the maximum number of links to a single file, (LINK_MAX), would be exceeded.
   ENOSPC  if the directory to contain the link cannot be extended.

SEE ALSO
   UNLINK(BA_OS).

LEVEL
   Level 1.
LOCKF(BA_OS)

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. 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_UNLOCK 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's 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.
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_UNLOCK and there are not enough entries in the system lock table to honor the request.
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:

```c
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
        */
```

File-locking and 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].

Enforcement-mode file and record locking will be added:

Sections of a file will be locked with advisory-mode or enforcement-mode locks depending on the mode of the file [see CHMOD(BA_OS)]

LEVEL
Level 1.
NAME
lseek — move read/write file-pointer

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

DESCRIPTION
The function lseek sets the file-pointer associated with fildes as follows:

If whence is 0, the function lseek will set the file-pointer equal to offset bytes.
If whence is 1, the function lseek will set the file-pointer equal to its current location plus offset.
If whence is 2, the function lseek will set the file-pointer equal to the length of the file 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 0, 1, or 2.

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.
LSEEK(BA_OS)

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).

FUTURE DIRECTIONS
The <unistd.h> header file will define the following symbolic constants for the argument whence to the seek and lseek functions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEEK_SET</td>
<td>set file-pointer to offset.</td>
</tr>
<tr>
<td>SEEK_CUR</td>
<td>set file-pointer to current plus offset.</td>
</tr>
<tr>
<td>SEEK_END</td>
<td>set file-pointer to EOF plus offset.</td>
</tr>
</tbody>
</table>

LEVEL
Level 1.
NAME
malloc, free, realloc, calloc, mallopt, mallinfo — fast main memory allocator

SYNOPSIS
#include <malloc.h>
char *malloc(size)
unsigned size;
void free(ptr)
char *ptr;
char *realloc(ptr, size)
char *ptr;
unsigned size;
char *calloc(nelem, elsize)
unsigned nelem, elsize;
int mallopt(cmd, value)
int cmd, value;
struct mallinfo mallinfo()

DESCRIPTION
The function malloc and the function free provide a simple general-purpose memory allocation package.

The function malloc returns a pointer to a block of at least size bytes suitably aligned for any use.

The argument to the function free is a pointer to a block previously allocated by the function malloc; after the function free is performed this space is made available for further allocation.

Undefined results will occur if the space assigned by the function malloc is overrun or if an invalid value for ptr is passed to the function free.

The function realloc changes the size of the block pointed to by ptr to size bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

The function calloc allocates space for an array of nelem elements of size elsize. The space is initialized to zeros.

Available in System V Release 2.0, the function mallopt plus the function mallinfo allow tuning the allocation algorithm at execution time.

The function mallopt initiates a mechanism that can be used to allocate small blocks of memory quickly. Using this scheme, a large-group (called a holding-block) of these small-blocks is allocated at one time. Then, each time a program requests a small amount of memory from malloc a pointer to one of the pre-allocated small-blocks is returned. Different holding-blocks are created for different sizes of small-blocks and are created when needed.
The function `malloc` allows the programmer to set three parameters to maximize efficient small-block allocation for a particular application. The three parameters are:

The value of `size` below which requests to `malloc` will be filled using the special small-block algorithm. Initially, this value, which will be called `maxfast`, is zero, which means that the small-block option is not normally in use by `malloc`.

The number of small-blocks in a holding-block. If holding-blocks have many more small-blocks than the program is using, space will be wasted. If holding-blocks are too small, have too few small-blocks in each, performance gain is lost.

The `grain` of small-block sizes. This value determines what range of small-block sizes will be considered to be the same size. This influences the number of separate holding-blocks allocated. For example, if `grain` were 16-bytes, all small-blocks of 16-bytes or less would belong to one holding-block and blocks from 17-bytes to 32-bytes would belong to another holding-block. Thus, if `grain` is too small space may be wasted because many holding-blocks may be created.

The values for the argument `cmd` to the function `malloc` are:

- **M_MXFAST**: Set `maxfast` to value. The algorithm allocates all blocks below the size of `maxfast` in large-groups and then doles them out very quickly. The default value for `maxfast` is 0.

- **M_NLBLKS**: Set `numblks` to value. The above mentioned large-groups each contain `numblks` blocks. The value for `numblks` must be greater than 1. The default value for `numblks` is 100.

- **M_GRAIN**: Set `grain` to value. The sizes of all blocks smaller than `maxfast` are considered to be rounded up to the nearest multiple of `grain`. The value for `grain` must be greater than 0. The default value for `grain` is the smallest number of bytes which will allow alignment of any data type. The value will be rounded up to a multiple of the default when `grain` is set.

- **M_KEEP**: Preserve data in a freed-block until the next call to the function `malloc`, `realloc`, or `calloc`. This option is provided only for compatibility with the older version of the function `malloc` and is not recommended.

These `cmd` values are defined by the `<malloc.h>` header file.

The function `malloc` may be called repeatedly, but the parameters may not be changed after the first small-block is allocated from a holding-block. If `malloc` is called again after the first small-block is allocated using the small-block algorithm, it will return an error.
The function `malloc` can be used during a program development to determine the best settings of these parameters for a particular application. The function `malloc` must not be called until after some storage has been allocated using the function `malloc`. The function `malloc` provides information describing space usage. It returns the structure `mallocinfo`, which includes the following members:

```c
int arena;       /* total space in arena */
int ordblks;     /* number of ordinary-blocks */
int smbioks;     /* number of small-blocks */
int hblkhd;      /* space in holding-block overhead */
int hblks;       /* number of holding-blocks */
int usmblks;     /* space in small-blocks in use */
int fsmblks;     /* space in free small-blocks */
int uordblks;    /* space in ordinary-blocks in use */
int fordblks;    /* space in free ordinary-blocks */
int keepcost;    /* space penalty for keep option */
```

The structure `mallocinfo` is defined by the `<malloc.h>` header file.

**RETURN VALUE**

Each of the allocation functions `malloc`, `realloc`, and `calloc` returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

The functions `malloc`, `realloc`, and `calloc` return a NULL pointer if `nbytes` is 0 or if there is not enough available memory. When the function `realloc` returns NULL, the block pointed to by `ptr` is left intact.

If the function `mallocopt` is called after any allocation from a holding-block or if the arguments `cmd` or `value` are invalid, the function `mallocopt` will return a non-zero value; otherwise, it will return 0.

**APPLICATION USAGE**

The functions `mallocopt` and `mallocinfo` and the `<malloc.h>` header file first appeared in System V Release 2.0.

In System V Release 2.0, the developer can control whether the contents of the freed space are destroyed or left undisturbed (see the function `mallocopt` above). In System V Release 1.0, the contents are left undisturbed.

Allocation time increases when many objects have been allocated and not freed. The additional System V Release 2.0 routines provide some flexibility in dealing with this.

**LEVEL**

Level 1.
MKNOD(BA_OS)

NAME
mknod — make a directory, or a special or ordinary-file

SYNOPSIS
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. Where the value of mode is interpreted as follows:

0170000 file type; one of the following:
  0010000 FIFO-special
  0020000 character-special
  0040000 directory
  0060000 block-special
  0100000 or 0000000 ordinary-file

0004000 set user-ID on execution
0002000 set group-ID on execution
0001000 (reserved)
0000777 access permissions; constructed from the following:
  0000400 read by owner
  0000200 write by owner
  0001000 execute (search on directory) by owner
  0000700 read, write, execute (search) by group
  0000077 read, write, 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's file mode creation mask: the function mknod clears each bit whose corresponding bit in the process's 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_dev 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.
- `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.

SEE ALSO
- `CHMOD(BA_OS)`, `EXEC(BA_OS)`, `STAT(BA_OS)`, `UMASK(BA_OS)`.

LEVEL
Level 1.
NAME
  mount — mount a file system

SYNOPSIS
  int mount(spec, dir, rwflag)
  char *spec, *dir;
  int rwflag;

DESCRIPTION
  The function mount requests that a removable file system contained on the
  block-special file identified by the argument spec be mounted on the directory identified by the argument dir.

  The arguments spec and dir are pointers to path-names.

  When the function mount succeeds, references to the file named by dir
  will refer to the root-directory on the mounted file system.

  The low-order bit of the argument rwflag is used to control write permission on the mounted file system; if the bit is set to 1, writing is forbidden; otherwise, writing is permitted according to individual file accessibility.

  The function mount may be invoked only by the super-user.

RETURN VALUE
  If successful, the function mount will return 0; otherwise, it will return -1 and errno will indicate the error.

ERRORS
  Under the following conditions, the function mount will fail and will set errno to:
  
  EPERM if the effective-user-ID is not super-user.
  ENOENT if any of the named files does not exist.
  ENOTDIR if a component of a path-prefix is not a directory.
  ENOTBLK if the device identified by spec is not block-special.
  ENXIO if the device identified by spec does not exist.
  ENOTDIR if dir is not a directory.
  EBUSY if dir is currently mounted on, is someone’s current working directory, or is otherwise busy.
  EBUSY if the device identified by spec is currently mounted.
  EBUSY if there are no more mount-table entries.

APPLICATION USAGE
  The function mount is not recommended for use by application-programs.

SEE ALSO
  U_MOUNT(BA_OS).
FUTURE DIRECTIONS

The external variable `errno` will be set to `EAGAIN` rather than `EBUSY` when the system mount-table is full.

Additional optional arguments will be added to the `mount` function. New bit-patterns will be added to the set of possible values of the argument `rwflag`. Some of these patterns will be used to indicate if an optional argument is present.

LEVEL

Level 1.
OPEN(BA_OS)

NAME
open — open 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 may affect subsequent reads and writes [see READ(BA_OS) and WRITE(BA_OS)].

When opening a FIFO with O_RDONLY or O_WRONLY set:
If O_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.
If the file exists, this flag has no effect. Otherwise, the file 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's 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.

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

If O_EXCL and O_CREAT are set, the function open will fail if the file exists.

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)].

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

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, or a component of the path-name should exist but does not.
- EACCES if a component of the path-prefix denies search permission; or if 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.
OPEN(BA_OS)

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.

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).

FUTURE DIRECTIONS
Enforcement-mode file and record-locking features will be added:

The function open will set errno to EAGAIN if the file exists, enforcement-mode file and record-locking is set and there are outstanding record-locks on the file [see CHMOD(BA_OS)].

LEVEL
Level 1.
NAME
    pause — suspend process until signal

SYNOPSIS
    int pause();

DESCRIPTION
    The function `pause` suspends the calling-process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling-process.

RETURN VALUE
    If the signal causes termination of the calling-process, the function `pause` will not return. In case of error, the function `pause` will return `-1` and `errno` will be set to `EINTR`.

ERRORS
    Under the following conditions, the function `pause` will fail and will set `errno` to:

    EINTR if the signal is caught by the calling-process and control is returned from the signal-catch function, the calling-process resumes execution from the point of suspension.

SEE ALSO
    `ALARM(BA_OS)`, `KILL(BA_OS)`, `SIGNAL(BA_OS)`, `WAIT(BA_OS)`.

LEVEL
    Level 1.
PIPE(BA_OS)

NAME
pipe — create an interprocess channel

SYNOPSIS
int pipe(fildes)
int fildes[2];

DESCRIPTION
The function pipe creates an I/O mechanism called a pipe and returns two file-descriptors, fildes[0] and fildes[1]. The file associated with fildes[0] is opened for reading, the file associated with fildes[1] is opened for writing, and the O_NDELAY flag is cleared.

Up to {PIPE_MAX} bytes of data are buffered by the pipe before the writing-process is blocked. A read-only file-descriptor fildes[0] accesses the data written to fildes[1] on a first-in-first-out, FIFO, basis.

RETURN VALUE
If successful, the function pipe will return 0; otherwise, it will return -1 and errno will indicate the error.

ERRORS
Under the following conditions, the function pipe will fail and will set errno to:

EMFILE if (OPEN_MAX)-1 or more file-descriptors are currently open for this process.

ENFILE if more than {SYS_OPEN} files would be open in the system.

SEE ALSO
READ(BA_OS), WRITE(BA_OS).

LEVEL
Level 1.
NAME
popen, pclose — initiate pipe to/from a process

SYNOPSIS
#include <stdio.h>
FILE *popen(command, type)
char *command, *type;
int pclose(stream)
FILE *stream;

DESCRIPTION
The function open creates a pipe between the calling program and the
command to be executed.

The arguments to open are pointers to null-terminated strings containing,
respectively, a command line [see SYSTEM(BA_OS)] and an I/O mode, either
r for reading or w for writing.

The function open returns a stream pointer such that one can write to the
standard input of the command, if the I/O mode is w by writing to the file
stream; and one can read from the standard output of the command, if the
I/O mode is r by reading from the file stream.

A stream opened by the function open should be closed by the function
pclose, which waits for the associated process to terminate and returns the
exit status of the command.

Because open files are shared, a type r command may be used as an input
filter and a type w command as an output filter.

RETURN VALUE
If files or processes cannot be created or if the command cannot be exe-
cuted, the function open will return a NULL pointer.

If stream is not associated with a opened command, the function
pclose will return -1.

APPLICATION USAGE
The FSEEK(BA_OS) routine should not be used with a stream opened by the
function open.

SEE ALSO
FCLOSE(BA_OS), FOPEN(BA_OS), PIPE(BA_OS), SYSTEM(BA_OS), WAIT(BA_OS).

LEVEL
Level 1.
READ(BA_OS)

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 asso­
ciated 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 empty pipe (or FIFO):

If the pipe is no longer open for writing, 0 will be returned indicating
end-of-file.

If _O_NDELAY is clear, the read will block until data is written to the
file or the file is no longer open for writing.

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.

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.
READ(BA_OS)

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.

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.

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.

Enforcement-mode file and record-locking will be added:

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

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 the blocking write-lock is removed.

The function read will fail and will set errno to:

EAGAIN if enforcement-mode file-locking and record-locking was set, O_NDELAY was set, and there was a blocking write-lock.
ENOLCK if the system record-lock table was full, so the read could not go to sleep until the blocking write-lock was removed.
READ(BA_OS)

LEVEL
  Level 1.
NAME
    setpgrp — set process-group-ID

SYNOPSIS
    int setpgrp()

DESCRIPTION
    The function setpgrp sets the process-group-ID of the calling-process to
    the process-ID of the calling process and returns the new process-group-ID.

RETURN VALUE
    If successful, the function setpgrp will return the new process-group-ID.

SEE ALSO
    EXEC(BA_OS), FORK(BA_OS), GETPID(BA_OS), KILL(BA_OS), SIGNAL(BA_OS).

LEVEL
    Level 1.
**SETUID(BA_OS)**

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

**SYNOPSIS**

```c
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 and effective-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`.

**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>
int (*signal(sig, func))()
int sig;
int (*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)*
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
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 default action for these signals is an abnormal process termination. See SIG_DFL.
The argument \texttt{func} is assigned one of three values: \texttt{SIG_DFL}, \texttt{SIG_IGN}, or an \textit{address} of a signal-catching function. The following actions are prescribed by these values:

**SIG_DFL** Terminate process upon receipt of a signal.

Upon receipt of the signal \texttt{sig}, the receiving process is to be terminated with all of the consequences outlined in \texttt{EXIT(BA_OS)}. In addition, if \texttt{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 \texttt{sig} is to be ignored.

\textbf{NOTE:} The signal \texttt{SIGKILL} cannot be ignored.

**address** Catch signal.

Upon receipt of the signal \texttt{sig}, the receiving process is to execute the signal-catching function pointed to by \texttt{func}. The signal number \texttt{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 \texttt{func} for the caught signal will be set to \texttt{SIG_DFL} unless the signal is \texttt{SIGILL}, or \texttt{SIGTRAP}.

The function \texttt{signal} will not catch an invalid function argument, \texttt{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 a \texttt{READ(BA_OS)}, \texttt{WRITE(BA_OS)}, \texttt{OPEN(BA_OS)}, or \texttt{IOCTL(BA_OS)} routine on a slow device (such as a terminal); or occurs during a \texttt{PAUSE(BA_OS)} routine; or occurs during a \texttt{WAIT(BA_OS)} routine that does not return immediately, the signal-catching function will be executed and then the interrupted routine may return a \texttt{-1} to the calling-process with \texttt{errno} set to \texttt{EINTR}.

\textbf{NOTE:} The signal \texttt{SIGKILL} cannot be caught.

A call to the function \texttt{signal} cancels a pending signal \texttt{sig} except for a pending \texttt{SIGKILL} signal.
SIGNAL(BA_OS)

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
(int(*)(()))-1 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 pro-
cess) 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
SIGABRT will be added to the <signal.h> header file [see
ABORT(BA_OS)].

A macro SIG_ERR will be defined by the <signal.h> header file to
represent the return value (int(*)(()))-1 of the function signal in
case of error.

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

In keeping with the proposed ANSI X3J11 standard, the argument func will
be declared as type pointer to a function returning void.

The following functions will be added to enhance the signal facility: sig-
set, sighold, sigrelse, sigignore and sigpause. These
functions will give a calling-process control over the disposition of a specified
signal that follows a signal that has been caught. When a signal has been
captured, the system will hold (defer) a succeeding signal of the type specified
should it occur. Similarly, processes will be able to establish critical regions
of code where an incoming-signal is deferred so the critical region can be
executed without losing the signal. Finally, a calling process will be able to
suspend if a specified signal has not yet occurred.

LEVEL
Level 1.
NAME
sleep — suspend execution for interval

SYNOPSIS
unsigned sleep(seconds)
unsigned seconds;

DESCRIPTION
The function sleep suspends the current-process from execution for the
number of seconds specified by the argument seconds. The actual
suspension-time may be less than that requested for two reasons: (1) Because
scheduled wakeups occur at fixed 1-second intervals (on the second, accord­
ing to an internal clock) and (2) because any signal caught will terminate the
sleep following execution of that signal-catching routine. Also, the
suspension-time may be longer than requested by an arbitrary amount due to
the scheduling of other activity in the system.

The function sleep sets an alarm signal and pauses until it (or some other
signal) occurs. The previous state of the alarm signal is saved and restored.
The calling-process may have set up an alarm signal before calling the func­
tion sleep. If the argument seconds exceeds the time until such an
alarm signal would occur, the process sleeps only until the alarm signal would
have occurred. The alarm signal-catching routine of the calling-process is
executed just before the function sleep returns. But if the suspension­
time is less than the time till such alarm, the prior alarm time remains
unchanged.

RETURN VALUE
If successful, the function sleep will return the unslept amount (the
requested time minus the time actually slept) in case the caller had an alarm
set to go off earlier than the end of the requested suspension-time or prema­
ture arousal due to another caught signal; otherwise, the function sleep
will return 0.

SEE ALSO
ALARM(BA_OS), PAUSE(BA_OS), SIGNAL(BA_OS).

LEVEL
Level 1.
NAME
stat, fstat — get file status

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

int stat(path, buf)
char *path;
struct stat *buf;

int fstat(fildes, buf)
int fildes;
struct stat *buf;

DESCRIPTION
The function stat obtains information about the named file.

The argument path points to a path-name naming a file. Neither read,
write, nor execute permission of the named file is required, but all directories
listed in the path-name leading to the file must be searchable.

Similarly, the function fstat obtains information about an open file asso­
ciated with the file-descriptor fildes [see file-descriptor in Chapter 4 —
Definitions].

The argument buf is a pointer to a structure stat into which informa­
tion is placed concerning the file.

The contents of the structure stat pointed to by buf include the follow­
ing members:

ushort st_mode;      /* file mode */
ino_t st_ino;        /* i-node number */
dev_t st_dev;         /* file-system identifier */
dev_t st_rdev;        /* device identifier, only */
                      /* for character-special */
                      /* or block-special files */
short st_nlink;      /* number of links */
ushort st_uid;        /* file owner user-ID */
ushort st_gid;        /* file group user-ID */
off_t st_size;        /* file size in bytes */
time_t st_atime;      /* time data last accessed */
time_t st_mtime;      /* time data last modified */
time_t st_ctime;      /* time file status last */
                      /* changed, in seconds since */
                      /* 00:00:00 GMT 1 Jan 70 */
**STAT(BA_OS)**

**st_mode** This field is the mode of the file as described in the MKNOD(BA_OS) routine.

**st_ino** This field uniquely identifies the file in a given file-system. The pair of fields st_ino and st_dev uniquely identifies ordinary-files.

**st_dev** This field uniquely identifies the file-system that contains the file. The value of the field may be used as input to the USTAT(BA_OS) routine to determine more information about this file-system. No other significance is associated with this value.

**st_rdev** This field should not be used by application-programs. The field is valid only for block-special or character-special files and only has significance on the system where the file was configured.

**st_nlink** This field should not be used by application-programs.

**st_size** For ordinary-files, this field is the address of the end of the file. For pipes or FIFOs, this field is the count of the data currently in the file. For block-special or character-special files, this field is not defined.

**st_atime** This field is the time when file-data was last accessed. The CREATE(BA_OS), LOCKF(BA_OS), MKNOD(BA_OS), PIPE(BA_OS), UTIME(BA_OS), and READ(BA_OS) routines change this field.

**st_mtime** This field is the time when file-data was last modified. The CREATE(BA_OS), MKNOD(BA_OS), PIPE(BA_OS), UTIME(BA_OS), and WRITE(BA_OS) routines change this field.

**st_ctime** This field is the time when file status was last changed. The CHMOD(BA_OS), CHOWN(BA_OS), CREATE(BA_OS), LINK(BA_OS), MKNOD(BA_OS), PIPE(BA_OS), UNLINK(BA_OS), UTIME(BA_OS), and WRITE(BA_OS) routines change this field.

The types ushort, ino_t, time_t, dev_t, and off_t are defined by the <sys/types.h> header file.

**RETURN VALUE**

If successful, both the function stat and the function fstat will return 0. Otherwise, both the function stat and the function fstat will return -1 and errno will indicate the error.
STAT(BA_OS)

ERRORS
Under the following conditions, the function stat 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.
EACCES if a component of the path-prefix denies search permission.
Under the following conditions, the function fstat will fail and will set errno to:
EBADF if the argument fildes is not a valid open file-descriptor.

SEE ALSO
CHMOD(BA_OS), CHOWN(BA_OS), CREAT(BA_OS), LINK(BA_OS), MKNOD(BA_OS),
PIPE(BA_OS), READ(BA_OS), TIME(BA_OS), UNLINK(BA_OS), UTIME(BA_OS),
WRITE(BA_OS).

LEVEL
Level 1.
NAME
stime — set time

SYNOPSIS
int stime(tp)
long *tp;

DESCRIPTION
The function stime sets the system time and date. The argument tp points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

RETURN VALUE
If successful, the function stime will return 0; otherwise, it will return -1 and errno will indicate the error.

ERRORS
Under the following conditions, the function stime will fail and will set errno to:
EPERM if the effective-user-ID of the calling-process is not super-user.

SEE ALSO
TIME(BA_OS).

LEVEL
Level 1.
SYNC(BA_OS)

NAME
sync — update super-block

SYNOPSIS
void sync()

DESCRIPTION
The function sync causes all information in transient memory that updates a file-system to be written out to the file-system. This includes modified super-blocks, modified i-nodes, and delayed block I/O.

The function sync should be used by programs which examine a file-system.

The writing, although scheduled, is not necessarily complete upon return from the function sync.

APPLICATION USAGE
The function sync is not recommended for use by application-programs.

LEVEL
Level 1.
NAME
system — issue a command

SYNOPSIS
#include <stdio.h>
int system(string)
char *string;

DESCRIPTION
The function system causes the argument string to be given as input to a command interpreter and execution process. That is, the argument string is interpreted as a command, and then the command is executed.

Commands
A blank is a tab or a space.

A word is a sequence of characters excluding blanks.

A parameter name is a sequence of letters, digits, or underscores beginning with a letter or underscore. A parameter is a parameter name, a digit, or any of the characters ?, $, or _.

A simple-command is a sequence of non-blank words separated by blanks. The first word specifies the path-name or file-name of the command to be executed. Except as specified below, the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 [see EXEC(BA_OS)]. The value of a simple-command is its exit status if it terminates normally, or (octal) 200+status if it terminates abnormally [see WAIT(BA_OS)].

A pipeline is a sequence of two or more simple-commands separated by the character \\. The standard output of each simple-command (except the last simple-command in the sequence) is connected by a PIPE(BA_OS) routine to the standard input of the next simple-command. Each simple-command is run as a separate process; the command execution process waits for the last simple-command to terminate. The exit status of a pipeline is the exit status of the last command.

A command is either a simple-command or a list enclosed in parentheses: (list). Unless otherwise stated, the value returned by a command is that of the last simple-command executed in the command.

A list is a command or a pipeline or a sequence of commands and pipelines separated by the characters ; or & or the character-pairs && or ||. Of these, the characters ; and &, which have equal precedence, have a precedence lower than that of the character-pairs && and ||, which have equal precedence. A list may optionally be terminated by the characters ; or &.

A series of commands and/or pipelines separated by the character ; are executed sequentially, while commands and pipelines terminated by the character & are executed asynchronously.
The character-pairs `&&` or `!!` cause the command or pipeline following it to be executed only if the preceding pipeline returns a zero (non-zero) exit status. An arbitrarily long sequence of new-lines may appear in a list, instead of the character `;`, to delimit commands.

Comments
A word beginning with the character `#` causes that word and all the following characters up to a new-line to be ignored.

Command Substitution
The standard output from a command bracketed by grave-accents (the character `'`) may be used as part or all of a word; trailing new-lines are removed.

Parameter Substitution
The character `$` is used to introduce substitutable keyword-parameters.

$ \{ parameter \}

The value, if any, of the parameter is substituted. The braces are required only when parameter is followed by a letter, digit, or underscore that is not to be interpreted as part of its name.

Keyword-parameters (also known as variables) may be assigned values by writing:

\[ parameter-name = value \]

The following parameters are automatically set:

Parameter Description
? The decimal value returned by the last synchronously executed command in this call to system.
$ The process-number of this process.
! The process-number of the last background command invoked in this call to system.

The following parameters are used by the command execution process:

Parameter Description
HOME The initial working (home) directory, initially set from the 6th-field in the /etc/passwd file [see PASSWD(BA_ENV)].
PATH The search path for commands (see Execution below).

Blank Interpretation
After parameter and command substitution, the results of substitution are scanned for internal field separator characters (space, tab and new-line) and split into distinct arguments where such characters are found. Explicit null arguments ("" or ' ') are retained. Implicit null arguments (those resulting from parameters that have no values) are removed.
**File Name Generation**

Following substitution, each word in the command is scanned for the characters *, ?, and [ . If one of these characters appears the word is regarded as a *pattern*. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern, the word is left unchanged. The character . at the start of a file name or immediately following the character /, as well as the character / itself, must be matched explicitly.

**Parameter** | **Description**
--- | ---
* | Matches any string, including the null string.
? | Matches any single character.
[ ... ] | Matches any one of the enclosed characters.
  
A pair of characters separated by the character — matches any character lexically between the pair, inclusive. If the first character following the opening [ is the character ! any character not enclosed is matched.

**Quoting**

The following characters have special meaning and cause termination of a word unless enclosed in quotation marks as explained below:

; & ( )  | < > new-line space tab

A character may be quoted (i.e., made to stand for itself) by preceding it with the character \ . The character-pair \new-line is ignored. All characters enclosed between a pair of single quote marks (''), except a single quote, are quoted. Inside double quote marks (""), parameter and command substitution occurs and the character \ quotes the characters \, *, ", and $.

**Input/Output**

Before a command is executed, its input and output may be redirected using a special notation. The following may appear anywhere in a simple-command, or may precede or follow a command and are not passed on to the invoked command; substitution occurs before word or digit is used:

**Notation** | **Description**
--- | ---
<word | Use file word as standard input (file-descriptor 0).
>word | Use file word as standard output (file-descriptor 1). If the file does not exist it is created; otherwise, it is truncated to zero length.
>>word | Use file word as standard output. If the file exists, output is appended to it (by first seeking to the end-of-file); otherwise, the file is created.
<&digit | Use the file associated with file-descriptor digit as standard input. Similarly for the standard output using >&digit.
<&- | The standard input is closed. Similarly for the standard output using >&-.
If a digit precedes any of the above, the digit specifies the file-descriptor to be associated with the file (instead of the default 0 or 1). For example:

... 2>&1

associates file descriptor 2 with the file currently associated with file descriptor 1.

The order in which redirections are specified is significant. Redirections are evaluated left-to-right. For example:

... 1>xxx 2>&1

first associates file-descriptor 1 with file xxx. It associates file-descriptor 2 with the file associated with file-descriptor 1 (i.e., xxx). If the order of redirections were reversed, file-descriptor 2 would be associated with the terminal (assuming file-descriptor 1 had been) and file-descriptor 1 would be associated with file xxx.

If a command is followed by the character & the default standard input for the command is the empty file /dev/null. Otherwise, the environment for the execution of a command contains the file-descriptors of the invoking process as modified by input/output specifications.

Environment

The environment [see EXEC(BA_OS)] is a list of parameter name-value pairs passed to an executed program in the same way as a normal argument list. On invocation, the environment is scanned and a parameter is created for each name found, giving it the corresponding value.

The environment for any simple-command may be augmented by prefixing it with one or more assignments to parameters. For example:

TERM=450 cmd;

Signals

The SIGINT and SIGQUIT signals for an invoked command are ignored if the command is followed by the character &; otherwise signals have the values inherited by the command execution process from its parent.

Execution

The above substitutions are carried out each time a command is executed. A new process is created and an attempt is made to execute the command via the EXEC(BA_OS) routines.

The parameter PATH defines the search path for the directory containing the command. The character : separates path-names. The default path is :/bin:/usr/bin (specifying the current directory, /bin, and /usr/bin, in that order). NOTE: The current directory is specified by a null path-name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path-list. If the command name contains the character / the search path is not used. Otherwise, each directory in the path is searched for an executable file.
Conventionally, the function `system` has been implemented with the Bourne shell, `SH(BU_CMD)` [see Volume II: Part II — Basic Utilities Extension Definition: Chapter 5 — Commands and Utilities]. The current definition of the function `system` is not intended to preclude that or its implementation by another command interpreter that provides the minimum functionality described here. Of course, any implementation may provide a superset of the functionality described.

**RETURN VALUE**

If successful, the function `system` will return the exit status of the last simple-command executed. Errors, such as syntax errors, cause a non-zero return value and execution of the command is abandoned.

**FILES**

`/dev/null`

**APPLICATION USAGE**

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

**SEE ALSO**

`DUP(BA_OS)`, `EXEC(BA_OS)`, `FORK(BA_OS)`, `PIPE(BA_OS)`, `SIGNAL(BA_OS)`, `ULIMIT(BA_OS)`, `UMASK(BA_OS)`, `WAIT(BA_OS)`.

**LEVEL**

Level 1.
NAME  
  time — get time

SYNOPSIS  
  long time((long *) 0)
  long time(tloc)
  long *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

times — get process and child-process elapsed times

SYNOPSIS

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

long times(buffer)
struct tms *buffer;

DESCRIPTION

The function times fills the structure pointed to by the argument
buffer with time-accounting information. The action of the function
time is undefined if the argument buffer points to an illegal address.

The following are the contents of the structure tms, which is defined by the
<sys/times.h> header file to include:

time_t tms_utime;
time_t tms_stime;
time_t tms_cutime;
time_t tms_cstime;

This information comes from the calling-process and each of its terminated
child-processes for which it has executed a WAIT(BA_OS) routine. All times
are defined in units of 1/(CLK_TCK)'s of a second.

The value of tms_utime is the CPU time used while executing instruc­
tions in the user-space of the calling-process.

The value of tms_stime is the CPU time used by the system on behalf of
the calling-process.

The value of tms_cutime is the sum of the tms_utime and
tms_cutime of the child-processes.

The value of tms_cstime is the sum of the tms_stime and
tms_cstime of the child-processes.

The type time_t is defined by the <sys/types.h> header file.

RETURN VALUE

If successful, the function times will return the elapsed real time, in units
of 1/(CLK_TCK)'s of a second, since an arbitrary point in the past (e.g., sys­
tem start-up time). This point does not change from one invocation of the
function times to another. When the function times fails, it will return
-1.

SEE ALSO

EXEC(BA_OS), FORK(BA_OS), TIME(BA_OS), WAIT(BA_OS).

LEVEL

Level 1.
NAME
ulimit — get and set user limits

SYNOPSIS
long ulimit(cmd, newlimit)
int cmd;
long newlimit;

DESCRIPTION
The function ulimit provides for control over process limits.

Values available for the argument cmd are:

1  Get the file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child-processes. Files of any size can be read.

2  Set the file size limit of the process equal to newlimit. Any process may decrease this limit, but only a process with an effective-user-ID of super-user may increase the limit.

RETURN VALUE
If successful, the function ulimit will return a non-negative value; otherwise, it will return -1, the limit will be unchanged and errno will indicate the error.

ERRORS
Under the following conditions, the function ulimit will fail and will set errno to:

EPERM  if a process with an effective-user-ID other than super-user attempts to increase its file size limit.

SEE ALSO
WRITE(BA_OS).

LEVEL
Level 1.
NAME
umask — set and get file creation mask

SYNOPSIS
int umask(cmask)
int cmask;

DESCRIPTION
The function umask sets the process's file mode creation mask [see
CREAT(BA_OS)] equal to the argument cmask and returns the previous
value of the mask. Only the owner, group, other permission bits of the argu-
ment cmask and the file mode creation mask are used.

RETURN VALUE
If successful, the function umask will return the previous value of the file
mode creation mask.

SEE ALSO
CHMOD(BA_OS), CREAT(BA_OS), MKNOD(BA_OS), OPEN(BA_OS).

LEVEL
Level 1.
UMOUNT(BA_OS)

NAME
umount — unmount a file system

SYNOPSIS
int umount(spec)
char *spec;

DESCRIPTION
The function umount requests that a previously mounted file system con­tained on the block-special device identified by the argument spec be unmounted.

The argument spec is a pointer to a path-name. After unmounting the file-system, the directory upon which the file-system was mounted reverts to its ordinary interpretation.

The function umount may be invoked only by the super-user.

RETURN VALUE
If successful, the function umount will return 0; otherwise, it will return -1 and errno will indicate the error.

ERRORS
Under the following conditions, the function umount will fail and will set errno to:
EPERM if the process's effective-user-ID is not super-user.
ENXIO if the device identified by spec does not exist.
ENOTDIR if a component of the path-prefix is not a directory.
ENOENT if the named file does not exist.
ENOTBLK if the device identified by spec is not block-special.
EINVAL if the device identified by spec is not mounted.
EBUSY if a file on the device identified by spec is busy.

APPLICATION USAGE
The function umount is not recommended for use by application-programs.

SEE ALSO
MOUNT(BA_OS).

LEVEL
Level 1.
NAME
uname — get name of current operating system

SYNOPSIS
#include <sys/utsname.h>

int uname(name)
struct utsname *name;

DESCRIPTION
The function uname stores information identifying the current operating system in the structure pointed to by the argument name.

The function uname uses the structure defined by the <sys/utsname.h> header file whose members include:

char sysname[SYS_NMLN];
char nodename[SYS_NMLN];
char release[SYS_NMLN];
char version[SYS_NMLN];
char machine[SYS_NMLN];

The function uname returns a null-terminated character string naming the current operating system in the character array sysname.

Similarly, the character array nodename contains the name that the system is known by on a communications network.

The members release and version further identify the operating system.

The member machine contains a standard name that identifies the hardware that the operating system is running on.

RETURN VALUE
If successful, the function uname will return a non-negative value; otherwise, it will return -1 and errno will indicate the error.

LEVEL
Level 1.
UNLINK(BA_OS)

NAME
unlink — remove directory entry

SYNOPSIS
int unlink(path)
char *path;

DESCRIPTION
The function unlink removes the directory entry named by the path-name pointed to by the argument path. When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, space occupied by the file is not released until all references to the file have been closed.

RETURN VALUE
If successful, the function unlink will return 0; otherwise, it will return -1 and errno will indicate the error.

ERRORS
Under the following conditions, the function unlink 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.
EACCES if a component of the path-prefix denies search permission.
EACCES if the directory containing the link to be removed denies write permission.
EPERM if the named file is a directory and the effective-user-ID of the process is not super-user.
EBUSY if the entry to be unlinked is the mount point for a mounted file system.
ETXTBSY if the entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed.
EROFS if the directory entry to be unlinked is part of a read-only file system.

SEE ALSO
CLOSE(BA_OS), LINK(BA_OS), OPEN(BA_OS).

LEVEL
Level 1.
NAME
ustat — get file system statistics

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

int ustat(dev, buf)
int 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 fol­
lowing 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
utime — set file access and modification times

SYNOPSIS
#include <sys/types.h>
int utime(path, times)
char *path;
struct utimbuf *times;

DESCRIPTION
The function utime sets the access and modification times of the named file.

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

If the argument times is NULL, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use the function utime in this manner.

If the argument times is not NULL, times is interpreted as a pointer to a structure utimbuf (see below) and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user may use the function utime this way.

The times in the structure utimbuf are measured in seconds since 00:00:00 GMT Jan. 1, 1970.

The structure utimbuf must be defined as:

```c
struct utimbuf {
    time_t actime; /* access time */
    time_t modtime; /* modification time */
};
```

The function utime will also cause the time of the last file status change (st_ctime) to be updated [see STAT(BA_OS)]. The type time_t is defined by the <sys/types.h> header file.

RETURN VALUE
If successful, the function utime will return 0; otherwise, it will return -1 and errno will indicate the error.

ERRORS
Under the following conditions, the function utime will fail and will set errno to:

ENOENT if the named file does not exist.
ENOTDIR if a component of the path-prefix is not a directory.
EACCES if a component of the path-prefix denies search permission.
EPERM if the effective-user-ID is not super-user and not the owner of the file and the argument times is not NULL.
UTIME(BA_OS)

EACCES if the effective-user-ID is not super-user and not the owner of the file and the argument times is NULL and write access is denied.

EROFS if the file-system containing the file is mounted read-only.

APPLICATION USAGE
The structure utimbuf must be declared by the application-program. The declaration is shown above.

SEE ALSO
STAT(BA_OS).

LEVEL
Level 1.
WAIT(BA_OS)

NAME
wait — wait for child-process to stop or terminate

SYNOPSIS
int wait(stat_loc)
int *stat_loc;
int wait((int *)0)

DESCRIPTION
The function wait suspends the calling-process until one of the immediate children terminates. If a child-process terminated prior to the call on the function wait, return is immediate.

If the argument stat_loc (taken as an integer) is non-zero, 16-bits of information called status are stored in the low-order 16-bits of the location pointed to by stat_loc. The status can be used to differentiate between stopped and terminated child-processes and if the child-process terminated, status identifies the cause of termination and passes useful information to the parent. This is accomplished in the following manner:

If the child-process terminated due to a call to the EXIT(BA_OS) routine, the low-order 8-bits of status will be zero and the next 8-bits will contain the low-order 8-bits of the argument that the child-process passed to the EXIT(BA_OS) routine.

If the child-process terminated due to a signal, the low-order 7-bits (i.e., bits 177) will contain the number of the signal that caused the termination. In addition, if abnormal process termination routines [see SIGNAL(BA_OS)] were successfully completed then the low-order eighth-bit (i.e., bit 200) will be set. The next 8-bits of status will be zero.

If a parent process terminates without waiting for its child-processes to terminate, a special system process inherits the child-processes [see EXIT(BA_OS)].

The function wait will fail and its actions are undefined if the argument stat_loc points to an illegal address.

RETURN VALUE
If the function wait returns due to the receipt of a signal, it will return -1 to the calling-process and will set errno to EINTR.

If the function wait returns due to a terminated child-process, it will return the process-ID of the child-process to the calling-process; otherwise, it will return immediately with a value of -1 and errno will indicate the error.

ERRORS
The function wait will fail and will set errno to:

ECHILD if the calling-process has no existing unwaited-for child-processes.
WAIT(BA_OS)

SEE ALSO
  EXEC(BA_OS), EXIT(BA_OS), FORK(BA_OS), PAUSE(BA_OS), SIGNAL(BA_OS).

LEVEL
  Level 1.
WRITE(BA_OS)

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.

If a write requests that more bytes be written than there is room for (e.g., beyond the user process's 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.

A write request of greater than PIPE_BUF bytes to a pipe (or FIFO) will behave differently.
If a write to a pipe (or FIFO) of more than \( \text{PIPE}_\text{BUF} \) bytes is requested, one of the following will occur:

If the \texttt{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 \( \text{PIPE}_\text{BUF} \) bytes may be interleaved on arbitrary byte boundaries with data written by other processes.

If the \texttt{O\_NDELAY} flag is set and the pipe is full, the process will not block and the function \texttt{write} will return \texttt{0}.

If the \texttt{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 \texttt{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 \( \text{PIPE}_\text{BUF} \) bytes, data from a write request of \( \text{PIPE}_\text{BUF} \) bytes or less will never be interleaved in the pipe with data from other processes.

**RETURN VALUE**

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

**ERRORS**

Under the following conditions, the function \texttt{write} will fail and will set \texttt{errno} to:

- \texttt{EBADF} if \texttt{fd} is not a valid file descriptor open for writing.
- \texttt{EPIPE} and \texttt{SIGPIPE} signal if an attempt is made to write to a pipe that is not open for reading by any process.
- \texttt{EFBIG} if an attempt was made to write a file that exceeds the process's file size limit or the system's maximum file size [see \texttt{ULIMIT(2)}].
- \texttt{EINTR} if a signal was caught during the write operation.
- \texttt{ENOSPC} if there is no free space remaining on the device containing the file.
- \texttt{EIO} if a physical I/O error has occurred.
- \texttt{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.
WRITE(BA_OS)

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.

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

FUTURE DIRECTIONS
Enforcement-mode file and record-locking will be added:

A write to an ordinary-file will be blocked if enforcement-mode file and record-locking is set, and there is a record-lock owned by another process on the segment of the file to be written.

If O_NDELAY is not set, the write will sleep until the blocking record-lock is removed.

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

EAGAIN if enforcement-mode file-locking and record-locking was set, O_NDELAY was set and there was a blocking record-lock.

EDEADLK if the write was going to sleep and cause a deadlock situation to occur.

ENOLCK if the system record-lock table was full, so the write could not go to sleep until the blocking record-lock was removed.

LEVEL
Level 1.
Chapter 7
General Library Routines
ABS(BA_LIB)

NAME
   abs — return integer absolute value

SYNOPSIS
   int abs(i)
   int i;

DESCRIPTION
   The function abs returns the absolute value of its integer operand.

APPLICATION USAGE
   In two-complement representation, the absolute value of the negative integer
   with largest magnitude (INT_MIN) is undefined. Some implementations may
   catch this as an error but others may ignore it.

SEE ALSO
   FLOOR(BA_LIB).

LEVEL
   Level 1.
NAME
j0, j1, jn, y0, y1, yn — Bessel functions

SYNOPSIS
#include <math.h>
double j0(x)
double x;
double j1(x)
double x;
double jn(n, x)
int n;
double x;
double y0(x)
double x;
double y1(x)
double x;
double yn(n, x)
int n;
double x;

DESCRIPTION
The functions j0 and j1 return Bessel functions of x of the first kind of orders 0 and 1 respectively.

The function jn returns the Bessel function of x of the first kind of order n.

The functions y0 and y1 return Bessel functions of x of the second kind of orders 0 and 1 respectively.

The function yn returns the Bessel function of x of the second kind of order n.

For the functions y0, y1 and yn, the argument x must be positive.

RETURN VALUE
Non-positive arguments cause y0, y1 and yn to return the value -HUGE and to set errno to EDOM. In addition, a message indicating argument DOMAIN error is printed on the standard error output.

Arguments too large in magnitude cause the functions j0, j1, y0 and y1 to return zero and to set errno to ERANGE. In addition, a message indicating TLOSS error is printed on the standard error output [see MATHERR(BA_LIB)].

APPLICATION USAGE
These error-handling procedures may be changed with the MATHERR(BA_LIB) routine.

Base System Definition Page 153
BESSEL(BA_LIB)

SEE ALSO
   MATHERR(BA_LIB).

LEVEL
   Level 1.
NAME
bsearch — binary search on a sorted table

SYNOPSIS
char *bsearch(key, base, nel, width, compar)
char *key;
char *base;
unsigned nel, width;
int (*compar)();

DESCRIPTION
The function bsearch is a binary search routine. It returns a pointer into
a table indicating where a datum may be found. The table must be previ­
ously sorted in increasing order according to a user-provided comparison
function, compar [see QSORT(BA_OS)].

The argument key points to a datum instance to be sought in the table.
The argument base points to the element at the base of the table.
The argument nel is the number of elements in the table.
The argument width is the size of an element in bytes.
The argument compar is the name of the comparison function, which is
called with two arguments of type char that point to the elements being
compared. The compar function must return an integer less than, equal to
or greater than zero, as the first argument is to be considered less than, equal
to or greater than the second.

RETURN VALUE
A NULL pointer is returned if the key cannot be found in the table.

APPLICATION USAGE
The pointers to the key and the element at the base of the table, key and
base, should be of type pointer-to-element and cast to type pointer-to­
character.

The comparison function need not compare every byte, so arbitrary data may
be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be
cast into type pointer-to-element.

EXAMPLE
The following example searches a table containing pointers to nodes consist­
ing of a string and its length. The table is ordered alphabetically on the
string in the node pointed to by each entry.
This code fragment reads in strings; it either finds the corresponding node and prints out the string and its length or it prints an error message.

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

#define TABSIZE 1000

struct node {
    char *string;
    int length;
};

struct node table[TABSIZE]; /* table to be searched */

struct node *node_ptr, node;

int node_compare(node1, node2)
struct node *node1, *node2;
{
    return strcmp(node1->string, node2->string);
}

This routine compares two nodes based on an alphabetical ordering of the string field.

SEE ALSO
HSEARCH(BA_LIB), LSEARCH(BA_LIB), QSORT(BA_LIB), TSEARCH(BA_LIB).

LEVEL
Level 1.
```
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) 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), SYSTEM(BA_OS).

LEVEL
   Level 1.
CONV(BA_LIB)

NAME
toupper, tolower, _toupper, _tolower, toascii — translate characters

SYNOPSIS
#include <ctype.h>
int toupper(c)
    int c;
int tolower(c)
    int c;
int _toupper(c)
    int c;
int _tolower(c)
    int c;
int toascii(c)
    int c;

DESCRIPTION
The functions toupper and tolower have as domain the range of the GETC(BA_LIB) routine: the integers from -1 through 255. If the argument of toupper represents a lower-case letter, the result is the corresponding upper-case letter. If the argument of tolower represents an upper-case letter, the result is the corresponding lower-case letter. All other arguments in the domain are returned unchanged.

The macros _toupper, _tolower, and _toascii are defined by the <ctype.h> header file. The macros _toupper and _tolower accomplish the same thing as toupper and tolower but have restricted domains and are faster. The macro _toupper requires a lower-case letter as its argument; its result is the corresponding upper-case letter. The macro _tolower requires an upper-case letter as its argument; its result is the corresponding lower-case letter. Arguments outside the domain cause undefined results.

The macro toascii yields its argument with all bits turned off that are not part of a standard ASCII character; it is intended for compatibility with other systems.

SEE ALSO
CTYPE(BA_LIB), GETC(BA_LIB).

LEVEL
Level 1.
NAME
crypt, setkey, encrypt — generate string encoding

SYNOPSIS
char *crypt(key, salt)
char *key, *salt;

void setkey(key)
char *key;

void encrypt(block, edflag)
char *block;
int edflag;

DESCRIPTION
The function crypt is a string-encoding function.

The argument key is a string to be encoded. The argument salt is a
two-character string chosen from the set [a-zA-Z0-9.]; this string is
used to perturb the encoding algorithm, after which the string that key
points to is used as the key to repeatedly encode a constant string. The
returned value points to the encoded string. The first two characters are the
salt itself.

The functions setkey and encrypt provide (rather primitive) access to
the encoding algorithm. The argument to the entry setkey is a character
array of length 64 containing only the characters with numerical value 0
and 1. If this string is divided into groups of 8, the low-order bit in each
group is ignored; this gives a 56-bit key. This is the key that will be used
with the above mentioned algorithm to encode the string block with the
function encrypt.

The argument to the entry encrypt is a character
array of length 64 con­tain­ing only the characters with numerical value 0
and 1. The argument
array is modified in place to a similar array representing the bits of the argu­
ment after having been subjected to the encoding algorithm using the key set
by setkey.

If the argument edflag is zero, the argument is encoded.

APPLICATION USAGE
The return value of the function crypt points to static data that are
overwritten by each call.

LEVEL
Level 1.

Optional: the functions crypt, setkey and encrypt may not be
present in all implementations of the Base System.
CTERMID(BA_LIB)

NAME
ctermid — generate file name for terminal

SYNOPSIS
#include <stdio.h>
char *ctermid(s)
    char *s;

DESCRIPTION
The function ctermid generates the path-name of the controlling terminal for the current process and stores it in a string.

If the argument s is a NULL pointer, the string is stored in an internal static area which will be overwritten at the next call to ctermid. The address of the static area is returned. Otherwise, s is assumed to point to a character array of at least L_ctermid elements; the path name is placed in this array and the value of s is returned. The constant L_ctermid is defined by the <stdio.h> header file.

APPLICATION USAGE
The difference between the TTYNAME(BA_LIB) routine and the function ctermid is that the TTYNAME(BA_LIB) routine must be passed a file-descriptor and returns the name of the terminal associated with that file-descriptor, while the function ctermid returns a string (e.g., /dev/tty) that will refer to the terminal if used as a file-name. Thus the TTYNAME(BA_LIB) routine is useful only if the process already has at least one file open to a terminal.

SEE ALSO
TTYNAME(BA_LIB).

LEVEL
Level 1.
NAME
ctime, localtime, gmtime, asctime, tzset — convert date and time to string

SYNOPSIS
#include <time.h>
char *ctime(clock)
long *clock;
struct tm *localtime(clock)
long *clock;
struct tm *gmtime(clock)
long *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 long integer, 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 Savings 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:

```c
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 Savings 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 Savings 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

```c
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 argument clock to the functions ctime, localtime and
gmtime will be defined by the <sys/types.h> header file as pointer
to time_t.

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

LEVEL
Level 1.
**NAME**

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii — classify characters

**SYNOPSIS**

```c
#include <ctype.h>

int isalpha(c)
int c;

int isupper(c)
int c;

int islower(c)
int c;

int isdigit(c)
int c;

int isxdigit(c)
int c;

int isalnum(c)
int c;

int isspace(c)
int c;

int ispunct(c)
int c;

int isprint(c)
int c;

int isgraph(c)
int c;

int iscntrl(c)
int c;

int isascii(c)
int c;
```

**DESCRIPTION**

These macros, which are defined by the `<ctype.h>` header file, classify character-coded integer values. Each is a predicate returning non-zero for true, zero for false. The function **isascii** is defined on all integer values; the rest are defined only where **isascii** is true and on the single non-ASCII value **EOF**, which is defined by the `<stdio.h>` header file and represents end-of-file.

- **isalpha**  
  c is a letter.

- **isupper**  
  c is an upper-case letter.

- **islower**  
  c is a lower-case letter.

- **isdigit**  
  c is a digit [0-9].
isxdigit  c is a hexadecimal digit [0–9], [A–F] or [a–f].
isalnum  c is an alphanumeric (letter or digit).
ispace  c is a space, tab, carriage-return, new-line, vertical-tab or form-feed.
ispunct  c is a punctuation mark (neither control nor alpha-numeric nor space).
isprint  c is a printing character, ASCII code 040 (space) through 0176 (tilde).
isgraph  c is a printing character, like isprint except false for space.
iscntrl  c is a delete character (0177) or an ordinary control-character (less than 040).
isascii  c is an ASCII character, code between 0 and 0177 inclusive.

RETURN VALUE
If the argument to any of these macros is not in the domain of the function, the result is undefined.

SEE ALSO
FOPEN(BA_OS), ASCII character set in Chapter 4 — Definitions.

LEVEL
Level 1.
DRAND48(BA_LIB)

NAME
drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48,
lcong48 — generate uniformly distributed pseudo-random numbers

SYNOPSIS
double drand48()
double erand48(xsubi)
unsigned short *xsubi[3];
long lrand48()
long nrand48(xsubi)
unsigned short *xsubi[3];
long mrand48()
long jrand48(xsubi)
unsigned short *xsubi[3];
void srand48(seedval)
long *seedval;
unsigned short *seed48(seed16v)
unsigned short *seed16v[3];
void lcong48(param)
unsigned short *param[7];

DESCRIPTION
This family of functions generates pseudo-random numbers using the well­known linear congruential algorithm and 48-bit integer arithmetic.

Functions drand48 and erand48 return non-negative double-precision
floating-point values uniformly distributed over the interval [0.0,1.0).

Functions lrand48 and nrand48 return non-negative long integers uni­formly distributed over the interval [0,2^{31}).

Functions mrand48 and jrand48 return signed long integers uniformly
distributed over the interval [−2^{31},2^{31}).

Functions srand48, seed48 and lcong48 are initialization entry
points, one of which should be invoked before either drand48, lrand48
or mrand48 is called. (Although it is not recommended practice, constant
default initializer values will be supplied automatically if drand48,
lrand48 or mrand48 is called without a prior call to an initialization
entry point.) Functions erand48, nrand48 and jrand48 do not
require an initialization entry point to be called first.
All the routines work by generating a sequence of 48-bit integer values, \( X_i \), according to the linear congruential formula:

\[
X_{n+1} = (aX_n + c) \mod m \quad n \geq 0
\]

The parameter \( m = 2^{48} \); hence 48-bit integer arithmetic is performed. Unless \( l\text{cong48} \) has been invoked, the multiplier value \( a \) and the addend value \( c \) are given by:

\[
a = 5\text{DEECE66}D_{16} = 2736731631558 \\
c = B_{16} = 138
\]

The value returned by any of the functions \( \text{drand48} \), \( \text{erand48} \), \( \text{lrand48} \), \( \text{nrand48} \), \( \text{mrand48} \) or \( \text{jrand48} \) is computed by first generating the next 48-bit \( X_i \) in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of \( X_i \) and transformed into the returned value.

The functions \( \text{drand48} \), \( \text{lrand48} \) and \( \text{mrand48} \) store the last 48-bit \( X_i \) generated in an internal buffer; that is why they must be initialized prior to being invoked. The functions \( \text{erand48} \), \( \text{nrand48} \) and \( \text{jrand48} \) require the calling program to provide storage for the successive \( X_i \) values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of \( X_i \) into the array and pass it as an argument. By using different arguments, functions \( \text{erand48} \), \( \text{nrand48} \) and \( \text{jrand48} \) allow separate modules of a large program to generate several independent streams of pseudo-random numbers. In other words, the sequence of numbers in each stream will not depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function \( \text{srand48} \) sets the high-order 32-bits of \( X_i \) to the \text{LONG_BIT} bits contained in its argument. The low-order 16-bits of \( X_i \) are set to the arbitrary value 330E\text{16}.

The initializer function \( \text{seed48} \) sets the value of \( X_i \) to the 48-bit value specified in the argument array. In addition, the previous value of \( X_i \) is copied into a 48-bit internal buffer, used only by \( \text{seed48} \), and a pointer to this buffer is the value returned by \( \text{seed48} \).

The initialization function \( \text{lcong48} \) allows the user to specify the initial \( X_i \), the multiplier value \( a \) and the addend value \( c \). Argument array elements \( \text{param}[0-2] \) specify \( X_i \), \( \text{param}[3-5] \) specify the multiplier \( a \), and \( \text{param}[6] \) specifies the 16-bit addend \( c \). After \( \text{lcong48} \) has been called, a subsequent call to either \( \text{srand48} \) or \( \text{seed48} \) will restore the standard multiplier and addend values, \( a \) and \( c \), specified on the previous page.
DRAND48(BA_LIB)

APPLICATION USAGE
The pointer returned by seed48, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time. Use the pointer to get at and store the last $X_i$ value and then use this

SEE ALSO
RAND(BA_LIB).

LEVEL
Level 1.
NAME
erf, erfc — error function and complementary error function

SYNOPSIS
#include <math.h>

double erf(x)
double x;

double erfc(x)
double x;

DESCRIPTION
The function erf returns the error function of \( x \), defined as follows:

\[
\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt
\]

APPLICATION USAGE
The function erfc is provided because of the extreme loss of relative accuracy if \( \text{erf}(x) \) is called for large \( x \) and the result subtracted from 1.0.

SEE ALSO
EXP(BA_LIB).

LEVEL
Level 1.
NAME
exp, log, log10, pow, sqrt — exponential, logarithm, power, square root functions

SYNOPSIS
#include <math.h>
double exp(x)
double x;
double log(x)
double x;
double log10(x)
double x;
double pow(x, y)
double x, y;
double sqrt(x)
double x;

DESCRIPTION
The function exp returns $e^x$.
The function log returns the natural logarithm of $x$. The value of $x$ must be positive.
The function log10 returns the logarithm base ten of $x$. The value of $x$ must be positive.
The functions pow returns $x^y$. If $x$ is zero, $y$ must be positive. If $x$ is negative, $y$ must be an integer.
The function sqrt returns the non-negative square root of $x$. The value of $x$ may not be negative.

RETURN VALUE
The function exp returns HUGE when the correct value would overflow or 0 when the correct value would underflow and sets errno to ERANGE.
The functions log and log10 return -HUGE and set errno to EDOM when $x$ is non-positive. A message indicating DOMAIN error (or SING error when $x$ is 0) is printed on the standard error output.
The function pow returns 0 and sets errno to EDOM when $x$ is 0 and $y$ is non-positive, or when $x$ is negative and $y$ is not an integer. In these cases a message indicating DOMAIN error is printed on the standard error output. When the correct value for pow would overflow or underflow, pow returns ±HUGE or 0 respectively and sets errno to ERANGE.
The function sqrt returns 0 and sets errno to EDOM when $x$ is negative. A message indicating DOMAIN error is printed on the standard error output.
APPLICATION USAGE
These error-handling procedures may be changed with the MATHERR(BA_LIB) routine.

SEE ALSO
HYPOT(BA_LIB), MATHERR(BA_LIB), SINH(BA_LIB).

FUTURE DIRECTIONS
A macro HUGE_VAL will be defined by the <math.h> header file. This macro will call a function which will either return $+\infty$ on a system supporting the IEEE P754 standard or $+\text{MAXDOUBLE}$ on a system that does not support the IEEE P754 standard.

The function exp will return HUGE_VAL when the correct value overflows.

The functions log and log10 will return $-\text{HUGE}_\text{VAL}$ when $x$ is not positive.

The function sqrt will return $-0$ when the value of $x$ is $-0$.

The return value of pow will be negative HUGE_VAL when an illegal combination of input arguments is passed to pow.

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, zero if y is zero or if x/y would overflow. Otherwise the number is f with the same sign as x, such that x=iy+f for some integer i, and |f|<|y|.
The function fabs returns the absolute value of x, i.e., |x|.

SEE ALSO
ABS(BA_LIB).

FUTURE DIRECTIONS
The function fmod will return x if y is zero or if x/y would overflow.

LEVEL
Level 1.
NAME
frexp, ldexp, modf — manipulate parts of floating-point numbers

SYNOPSIS

double frexp(value, eptr)
double value;
int *eptr;

double ldexp(value, exp)
double value;
int exp;

double modf(value, iptr)
double value, *iptr;

DESCRIPTION
Every non-zero number can be written uniquely as $x \cdot 2^n$, where the mantissa
(fraction) $x$ is in the range $0.5 \leq |x| < 1.0$ and the exponent $n$ is an integer.
The function frexp returns the mantissa of a double value and
stores the exponent indirectly in the location pointed to by eptr. If
value is 0, both results returned by frexp are 0.

The function ldexp returns the quantity $value \cdot 2^{exp}$.

The function modf returns the fractional part of value and stores the
integral part indirectly in the location pointed to by iptr. Both the frac­
tional and integer parts have the same sign as value.

RETURN VALUE
If ldexp would cause overflow, ±HUGE is returned (according to the sign
of value) and errno is set to ERANGE.

If ldexp would cause underflow, 0 is returned and errno is set to
ERANGE.

FUTURE DIRECTIONS
A macro HUGE_VAL will be defined by the <math.h> header file. This
macro will call a function which will either return $+\infty$ on a system support­
ing the IEEE P754 standard or $+(\text{MAXDOUBLE})$ on a system that does not
support the IEEE P754 standard.

The return value of ldexp will be ±HUGE_VAL (according to the sign of
value) in case of overflow.

LEVEL
Level 1.
FTW(BA_LIB)

NAME

ftw — walk a file tree

SYNOPSIS

#include <ftw.h>

int ftw(path, fn, param)
char *path;
int (*fn)();
int param;

DESCRIPTION

The function ftw recursively descends the directory hierarchy rooted in path. For each object in the hierarchy, the function ftw calls a user-defined function fn passing it three arguments. The first argument passed is a character pointer to a null-terminated string containing the name of the object. The second argument passed to fn is a pointer to a stat structure [see STAT(BA_OS)] containing information about the object, and the third argument passed is an integer flag. Possible values of the flag, defined by the <ftw.h> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read and FTW_NS for an object for which stat could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the contents of the stat structure are undefined.

The function ftw visits a directory before visiting any of its descendants.

The function ftw uses one file-descriptor for each level in the tree. The argument param limits the number of file-descriptors so used. The argument param should be in the range of 1 to {OPEN_MAX}. The function ftw will run more quickly if param is at least as large as the number of levels in the tree.

RETURN VALUE

The tree traversal continues until the tree is exhausted, an invocation of fn returns a nonzero value or some error is detected within ftw (such as an I/O error). If the tree is exhausted, the function ftw returns 0. If the function fn returns a non-zero value, the function ftw stops its tree traversal and returns whatever value was returned by the function fn.

If the function ftw encounters an error other than EACCES (see FTW_DNR and FTW_NS above), it returns -1 and errno is set to the type of error. The external variable errno may contain the error values that are possible when a directory is opened [see OPEN(BA_OS)] or when the STAT(BA_OS) routine is executed on a directory or file.

APPLICATION USAGE

Because the function ftw is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.
SEE ALSO
    STAT(BA_OS), MALLOC(BA_OS).

LEVEL
    Level 1.
GAMMA(BA_LIB)

NAME
gamma — log gamma function

SYNOPSIS
#include <math.h>
double gamma(x)
double x;
extern int signgam;

DESCRIPTION
The function gamma returns ln(|Γ(x)|), where Γ(x) is defined as:
\[ \int_0^\infty e^{-t}t^{x-1}dt \]
The sign of Γ(x) is returned in the external integer signgam. The argument x may not be a non-positive integer.

The following C program fragment might be used to calculate Γ:

```c
if ( ((y = gamma(x)) > LN_MAXDOUBLE) )
    error();

y = signgam * exp(y);
```

RETURN VALUE
For non-positive integer arguments HUGE is returned, and errno is set to EDOM. A message indicating SING error is printed on the standard error output [see MATHERR(BA_LIB)].

If the correct value would overflow, gamma returns HUGE and sets errno to ERANGE.

APPLICATION USAGE
These error-handling procedures may be changed with the MATHERR(BA_LIB) routine.

SEE ALSO
EXP(BA_LIB), MATHERR(BA_LIB).

FUTURE DIRECTIONS
A macro HUGE_VAL will be defined by the <math.h> header file. This macro will call a function which will either return +\infty on a system supporting the IEEE P754 standard or +[MAXDOUBLE] on a system that does not support the IEEE P754 standard.

If the correct value overflows, gamma will return HUGE_VAL.

LEVEL
Level 1.
NAME
getc, getchar, fgetc, getw — get character or word from a stream

SYNOPSIS
#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar()

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;

DESCRIPTION
The function getc returns the next character (i.e., byte) from the named input stream as an integer. It also moves the file pointer, if defined, ahead one character in stream. The macro getchar is defined as getc(stdin). Both getc and getchar are macros.

The function fgetc behaves like getc but is a function rather than a macro. The function fgetc runs more slowly than getc but it takes less space per invocation and its name can be passed as an argument to a function.

The function getw returns the next word (i.e., integer) from the named input stream. The function getw increments the associated file pointer, if defined, to point to the next word. The size of a word is the size of an integer and varies from machine to machine. The function getw assumes no special alignment in the file.

RETURN VALUE
These functions return the constant EOF at end-of-file or upon an error. Because EOF is a valid integer, the FERROR(BA_OS) routine should be used to detect getw errors.

APPLICATION USAGE
If the integer value returned by getc, getchar or fgetc is stored into a character variable and then compared against the integer constant EOF, the comparison may never succeed because sign-extension of a character on widening to integer is machine-dependent.

Because of possible differences in word length and byte ordering, files written using putw are machine-dependent and may not be read using getw on a different processor.

Because it is implemented as a macro, getc treats incorrectly a stream argument with side effects. In particular, getc(*f++) does not work sensibly. The function fgetc should be used instead.
GETC(BA_LIB)

SEE ALSO
   FCLOSE(BA_OS), FERROR(BA_OS), FOPEN(BA_OS), FREAD(BA_OS),
   GETS(BA_LIB), PUTC(BA_LIB), SCANF(BA_LIB).

LEVEL
   Level 1.
NAME
  getenv — return value for environment name

SYNOPSIS
  char *getenv(name)
  char *name;

DESCRIPTION
  The function `getenv` searches the environment list for a string of the form
  `name = value` and returns a pointer to the `value` in the current
  environment if such a string is present. Otherwise a NULL pointer is
  returned.

SEE ALSO
  EXEC(BA_OS), SYSTEM(BA_OS), PUTENV(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 may or may not 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.

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 optstring. 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:

```c
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)) {
            ...
        }
    }
}
```

FUTURE DIRECTIONS

The function `getopt` will be enhanced to enforce all rules of the System V Command Syntax Standard (see below). All new System V commands will conform to the command syntax standard described here. Existing commands will migrate toward the new standard if they do not already meet it. Applications whose user-interface is command-like may want to be consistent with the syntax standard.
GETOPT(BA_LIB)

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.

LEVEL

Level 1.
NAME
gets, fgets — get a string from a stream

SYNOPSIS
#include <stdio.h>
char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
int n;
FILE *stream;

DESCRIPTION
The function gets reads characters from the standard input stream, stdin, into the array pointed to by s until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

The function fgets reads characters from the stream into the array pointed to by s until n-1 characters are read or a new-line character is read and transferred to s or an end-of-file condition is encountered. The string is then terminated with a null character.

RETURN VALUE
If end-of-file is encountered and no characters have been read, no characters are transferred to s and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise s is returned.

APPLICATION USAGE
Reading too long a line through gets may cause gets to fail. The use of fgets is recommended.

SEE ALSO
FERROR(BA_OS), FOPEN(BA_OS), FREAD(BA_OS), GETC(BA_LIB), SCANF(BA_LIB).

LEVEL
Level 1.
NAME
hsearch, hcreate, hdestroy — manage hash search tables

SYNOPSIS
#include <search.h>
ENTRY *hsearch(item, action)
ENTRY item;
ACTION action;
int hcreate(nel)
unsigned nel;
void hdestroy( )

DESCRIPTION
The function hsearch is a hash-table search routine. It returns a pointer
into a hash table indicating the location at which an entry can be found. The
comparison function used by hsearch is the function strcmp [see
STRING(BA_LIB)].

The argument item is a structure of type ENTRY (defined by the
<search.h> header file) containing two character pointers: item.key
pointing to the comparison key and item.data pointing to any other data
to be associated with that key. (Pointers to types other than char should
be cast to pointer-to-character.)

The argument action is a member of an enumeration type ACTION,
defined by the <search.h> header file, indicating the disposition of the
entry if it cannot be found in the table.

ENTER indicates that the item should be inserted in the table at an
appropriate point. Given a duplicate of an existing item, the new item is not
entered, and hsearch returns a pointer to the existing item.

FIND indicates that no entry should be made. Unsuccessful resolution is
indicated by the return of a NULL pointer.

The function hcreate allocates sufficient space for the table and must be
called before hsearch is used. The value of nel is an estimate of the
maximum number of entries that the table will contain. This number may be
adjusted upward by the algorithm in order to obtain certain mathematically
favorable circumstances.

The function hdestroy destroys the search table and may be followed by
another call to hcreate.

RETURN VALUE
The function hsearch returns a NULL pointer if either the action is
FIND and the item could not be found or the action is ENTER and the
table is full.

The function hcreate returns 0 if it cannot allocate sufficient space for
the table.
APPLICATION USAGE

The functions *hsearch* and *hcreate* use the MALLOC(BA_OS) routine to allocate space.

EXAMPLE

The example reads in strings followed by two numbers and stores them in a hash table. It then reads in strings and finds the entry in the table and prints it.

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

struct info { /* these are in the table */
    int age, room; /* apart from the key. */
};
#define NUM_EMPL 5000 /* # of elements in the table */

main()
{
    /* space for strings */
    char string_space[NUM_EMPL*20];
    /* space for employee info */
    struct info info_space[NUM_EMPL];
    /* next avail space for strings */
    char *str_ptr = string_space;
    /* next avail space for info */
    struct info *info_ptr = info_space;
    ENTRY item, *found_item, *hsearch();
    char name_to_find[30]; /* name to look for in table */
    int i = 0;
    /* create table */
    (void) hcreate(NUM_EMPL);
    while (scanf("%s%d%d", str_ptr, &info_ptr->age, &info_ptr->room) != EOF && i++ < NUM_EMPL) {
        /* put info in structure, and structure in item */
        item.key = str_ptr;
        item.data = (char *)info_ptr;
        str_ptr += strlen(str_ptr) + 1;
        info_ptr++;
        /* put item into table */
        (void) hsearch(item, ENTER);
    }
    /* access table */
    item.key = name_to_find;
    while (scanf("%s", item.key) != EOF) {
        if ((found_item = hsearch(item, FIND))) != NULL) {
            /* if item is in the table */
            (void) printf("found %s, age = %d, room = %d\n",
                found_item->key,
                ((struct info *)found_item->data)->age,
                ((struct info *)found_item->data)->room);
        } else {
            (void) printf("no such employee %s\n", name_to_find)
        }
    }
}
HSEARCH(BA_LIB)

SEE ALSO
MALLOCP(BA_OS), BSEARCH(BA_LIB), LSEARCH(BA_LIB), STRING(BA_LIB),
TSEARCH(BA_LIB).

FUTURE DIRECTIONS
The restriction of having only one hash search table active at any given time
will be removed.

LEVEL
Level 1.
NAME
hypot — Euclidean distance function

SYNOPSIS
#include <math.h>

double hypot(x, y)
double x, y;

DESCRIPTION
The function hypot returns sqrt(x * x + y * y), taking precautions against unwarranted overflows.

RETURN VALUE
When the correct value would overflow, hypot returns HUGE and sets errno to ERANGE.

These error-handling procedures may be changed with the function defined by the MATHERR(BA_LIB) routine.

SEE ALSO
MATHERR(BA_LIB).

FUTURE DIRECTIONS
A macro HUGE_VAL will be defined by the <math.h> header file. This macro will call a function which will either return +∞ on a system supporting the IEEE P754 standard or +(MAXDOUBLE) on a system that does not support the IEEE P754 standard.

The function hypot will return HUGE_VAL when the correct value overflows.

LEVEL
Level 1.
LSEARCH(BA_LIB)

NAME
lsearch, lfind — linear search and update

SYNOPSIS
#include <search.h>
char *lsearch(key, base, nelp, width, compar)
char *key;
char *base;
unsigned *nelp;
unsigned width;
int (*compar)();
char *lfind(key, base, nelp, width, compar)
char *key;
char *base;
unsigned *nelp;
unsigned width;
int (*compar)();

DESCRIPTION
The function lsearch is a linear search routine. It returns a pointer into a
table indicating where a datum may be found. If the datum does not occur,
it is added at the end of the table. The value of key points to the datum to
be sought in the table. The value of base points to the first element in the
table. The value of nelp points to an integer containing the current
number of elements in the table. The value of width is the size of an ele­
ment in bytes. The variable pointed to by nelp is incremented if the
datum is added to the table. The value of compar is the name of the com­
parison function which the user must supply (strcmp, for example). It is
called with two arguments that point to the elements being compared. The
function must return zero if the elements are equal and non-zero otherwise.

The function lfind is the same as lsearch except that if the datum is
not found, it is not added to the table. Instead, a NULL pointer is returned.

RETURN VALUE
If the searched for datum is found, both the functions lsearch and
lfind return a pointer to it. Otherwise, the function lfind returns
NULL and the function lsearch returns a pointer to the newly added ele­
ment.

APPLICATION USAGE
The function lfind was added to System V in System V Release 2.0.

The pointers to the key and the element at the base of the table should be of
type pointer-to-element and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may
be contained in the elements in addition to the values being compared.
Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

Space for the table must be managed by the application-program. Undefined results can occur if there is not enough room in the table to add a new item.

**EXAMPLE**

This fragment will read in \( \leq \) TABSIZE strings of length \( \leq \) ELSIZE and store them in a table, eliminating duplicates.

```c
#include <stdio.h>
#include <search.h>
#define TABSIZE 50
#define ELSIZE 120
char line[ELSIZE], tab[TABSIZE][ELSIZE], *lsearch();
unsigned nel = 0;
int strcmp();
... while (fgets(line, ELSIZE, stdin) != NULL &&
    nel < TABSIZE)
    (void) lsearch(line, (char *)tab, &nel,
                   ELSIZE, strcmp);
...```

**SEE ALSO**

BSEARCH(BA_LIB), HSEARCH(BA_LIB), TSEARCH(BA_LIB).

**FUTURE DIRECTIONS**

A NULL pointer will be returned by the function lsearch with errno set appropriately, if there is not enough room in the table to add a new item.

**LEVEL**

Level 1.
MATHERR(BA_LIB)

NAME
matherr — error-handling function

SYNOPSIS
#include <math.h>
int matherr(x)
struct exception *x;

DESCRIPTION
The function matherr is invoked by math library routines when errors are detected. Users may define their own procedures for handling errors, by including a function named matherr in their programs. The function matherr must be of the form described above. When an error occurs, a pointer to the exception structure x will be passed to the user-supplied matherr function. This structure, which is defined by the <math.h> header file, includes the following members:

    int type;
    char *name;
    double arg1, arg2, retval;

The element type is an integer describing the type of error that has occurred from the following list defined by the <math.h> header file:

- DOMAIN argument domain error.
- SING argument singularity.
- OVERFLOW overflow range error.
- UNDERFLOW underflow range error.
- TLOSS total loss of significance.
- PLOSS partial loss of significance.

The element name points to a string containing the name of the routine that incurred the error. The elements arg1 and arg2 are the first and second arguments with which the routine was invoked.

The element retval is set to the default value that will be returned by the routine unless the user's matherr function sets it to a different value.

If the user's matherr function returns non-zero, no error message will be printed, and errno will not be set.

If the function matherr is not supplied by the user, the default error-handling procedures, described with the math library routines involved, will be invoked upon error. These procedures are also summarized in the table below. In every case, errno is set to EDOM or ERANGE and the program continues.
ERRORS

DEFAULT ERROR HANDLING PROCEDURES

<table>
<thead>
<tr>
<th>type</th>
<th>DOMAIN</th>
<th>SING</th>
<th>OVERFLOW</th>
<th>UNDERFLOW</th>
<th>TLOSS</th>
<th>PLOSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>errno</td>
<td>EDOM</td>
<td>EDOM</td>
<td>RANGE</td>
<td>RANGE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BESSL:</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>M, 0</td>
<td>*</td>
</tr>
<tr>
<td>y0, y1, ym (arg &lt; M)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>M, 0</td>
<td>*</td>
</tr>
<tr>
<td>EXP:</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LOG, LOG10:</td>
<td>M, -H</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(arg &lt; 0)</td>
<td>(arg = 0)</td>
<td>-</td>
<td>M, -H</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>POW:</td>
<td>-</td>
<td>-</td>
<td>±H</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>neg ** non-int</td>
<td>M, 0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0 ** non-pos</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SQRT:</td>
<td>M, 0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GAMMA:</td>
<td>-</td>
<td>M, H</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>HYPOT:</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SINF:</td>
<td>-</td>
<td>-</td>
<td>±H</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>COSH:</td>
<td>-</td>
<td>-</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SIN, COS, TAN:</td>
<td>M, 0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>M, 0</td>
<td>*</td>
</tr>
<tr>
<td>ASIN, ACOS, ATAN2:</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

ABBREVIATIONS

- As much as possible of the value is returned.
- M Message is printed (EDOM error).
- H HUGE is returned.
- -H -HUGE is returned.
- ±H HUGE or -HUGE is returned.
- 0 0 is returned.

EXAMPLE

```c
#include <math.h>

int matherr(x)
register struct exception *x;
{
    switch (x->type) {
    case DOMAIN:
        /* change sqrt to return sqrt(-arg1), not 0 */
        if (!strcmp(x->name, "sqrt")) {
            x->retval = sqrt(-x->arg1);
            return (0); /* print message and set errno */
        }
        break;
    case SING:
        /* SING or other DOMAIN errs, print message and abort */
        fprintf(stderr, "domain error in \%s\n", x->name);
        abort();
        break;
    case PLOSS:
        /* print detailed error message */
        fprintf(stderr, "loss of significance in \%s(\%g) = \%g\n", 
            x->name, x->arg1, x->retval);
        return (1); /* take no other action */
    }
    return (0); /* all other errors, execute default procedure */
}
```

Base System Definition Page 191
MATHERR(BA_LIB)

FUTURE DIRECTIONS
The math functions which return HUGE or ±HUGE on overflow will return HUGE_VAL or ±HUGE_VAL respectively.

LEVEL
Level 1.
NAME
memccpy, memchr, memcmp, memcpy, memset — memory operations

SYNOPSIS
#include <memory.h>
char *memccpy(s1, s2, c, n)
char *s1, *s2;
int c, n;
char *memchr(s, c, n)
char *s;
int c, n;
int memcmp(s1, s2, n)
char *s1, *s2;
int n;
char *memcpy(s1, s2, n)
char *s1, *s2;
in n;
char *memset(s, c, n)
char *s;
int c, n;

DESCRIPTION
These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

The function memccpy copies characters from memory area s2 into s1, stopping after the first occurrence of character c has been copied or after n characters have been copied, whichever comes first. It returns a pointer to the character after the copy of c in s1, or a NULL pointer if c was not found in the first n characters of s2.

The function memchr returns a pointer to the first occurrence of character c in the first n characters of memory area s, or a NULL pointer if c does not occur.

The function memcmp compares its arguments, looking at the first n characters only. It 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 memcpy copies n characters from memory area s2 to s1. It returns s1.

The function memset sets the first n characters in memory area s to the value of character c. It returns s.
APPLICATION USAGE
All these functions are defined by the `<memory.h>` header file.

The function `memcmp` uses 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 be unpredictable.

SEE ALSO
`STRING(BA_LIB)`.

FUTURE DIRECTIONS
The declarations in the `<memory.h>` header file will be moved to the `<string.h>` header file.

LEVEL
Level 1.
NAME
mktemp — make a unique file name

SYNOPSIS
char *mktemp(template)
char *template;

DESCRIPTION
The function mktemp replaces the contents of the string pointed to by template by a unique file name and returns template. The string in template should look like a file name with six trailing Xs; mktemp will replace the Xs with a letter and the current process-ID. The letter will be chosen so that the resulting name does not duplicate an existing file.

RETURN VALUE
The function mktemp returns the pointer template. If a unique name cannot be created, template will point to a null string.

SEE ALSO
GETPID(BA_OS), TMPFILE(BA_LIB), TMPNAM(BA_LIB).

FUTURE DIRECTIONS
A NULL pointer will be returned if a unique name cannot be created.

LEVEL
Level 1.
NAME
 perror — system error messages

SYNOPSIS
 void perror(s)
  char *s;
  extern int errno;
  extern char *sys_errlist[];
  extern int sys_nerr;

DESCRIPTION
 The function perror produces a message on the standard error output
 describing the last error encountered during a call to a function.

 The string pointed to by the argument s is printed first, then a colon and a
 blank, then the message and a new-line. To be of most use, the argument
 string should include the name of the program that incurred the error.

 The error number is taken from the external variable errno, which is set
 when errors occur but not cleared when successful calls are made.

 If given a null-string, the function perror prints only the message and a
 new-line.

 The array of message strings sys_errlist is provided to make messages
 consistent. The variable errno can be used as an index in this array to get
 the message string without the new-line. The external variable sys_nerr
 is the largest message number provided for in the array; it should be checked
 because new error codes may be added to the system before they are added
 to the array.

FUTURE DIRECTIONS
 New error handling routines will be added to support the System V Error
 Message Standard as a tool for application-developers to use. The System V
 Error Message Standard is designed to apply to: firmware/diagnostics, the
 operating system, networks, System V commands, languages and, when
 appropriate, applications. All new System V error messages will follow the
 standard, and existing error messages will be modified over time. The stan­
 dard System V error message as seen by the end-user may have up to five
 informational elements:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LABEL</td>
<td>source of the error.</td>
</tr>
<tr>
<td>SEVERITY</td>
<td>one of at least 4 severity codes.</td>
</tr>
<tr>
<td>PROBLEM</td>
<td>description of the problem.</td>
</tr>
<tr>
<td>ACTION</td>
<td>error-recovery action.</td>
</tr>
<tr>
<td>TAG</td>
<td>unique error message identifier.</td>
</tr>
</tbody>
</table>

Each element is described in more detail below.
The standard specifies the information that is important in error recovery. It does not specify the format in which the information is delivered. For example, if a system had a graphical user interface, the LABEL might be presented as an icon. An operating system error message meeting the standard information requirements is shown below. Here, OS is the LABEL, HALT is the SEVERITY, Timeout Table Overflow is the PROBLEM, See Administration Manual is the ACTION, and OS-136 is the TAG.

OS: HALT: Timeout Table Overflow.
TO FIX: See Administration Manual. OS-136

The standard allows systematic omission of one or more elements in specific environments that do not need them for successful error recovery. For example, while operating system errors need all five elements, a firmware error message can omit the ACTION because an expert service person is typically the user of this message and the ACTION may be too long to store in firmware. Software that obviously puts the user in a special environment (e.g., a spread-sheet program) where the user will only see errors from that environment may omit the LABEL. Because a primary use of the TAG is for reporting or to point to on-line documentation, it may be omitted when appropriate (e.g., when there is no on-line documentation).

LABEL This element of the message identifies the error source (e.g., OS, UUCP, application-program-name, etc.) and could double as a pointer to documentation.

SEVERITY This element of the message indicates the consequences of the error for the user. Four levels of severity (which can be expanded by system builders who want additional distinctions) are outlined below.

HALT signifies that the processor, OS, application, or database is corrupted and that processing should be stopped immediately to rectify the problem. This severity signifies an emergency.

ERROR signifies that a condition that may soon interfere with resource use has occurred. This severity alerts the user that some corrective action is needed.

WARNING signifies an aberrant condition (e.g., stray hardware interrupt, free file space is low) that should be monitored, but requires no immediate action.

INFO simply provides some information about a user request or about the state of the system (e.g., a printer taken off-line).

PROBLEM This element of the message clearly describes the error condition. In much of today's software, this element is the only one provided in the message.
ACTION  This element of the message describes the first step to be taken in the error-recovery process. For OS errors, this section of the message might be one of five standard strings: See Hardware Vendor, See Software Vendor, See Administrator Procedure, See Operator Procedure, or See Manual. These strings should be clearly identified as action to be taken (e.g., by preceding them with the prefix: TO FIX:).

TAG  This is a unique identifier for the message, used both internally and to obtain online documentation for the message on those systems that have capacity to store such information.

LEVEL  
Level 2: January 1, 1985.
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, 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 and f conversions, the maximum number of significant digits for the 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.

An optional 1 (ell) to specify that a following d, o, u, x or X conversion character applies to a long integer arg. An 1 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 args specifying field width or precision must appear before the arg (if any) to be converted.
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, 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,o,u,x,X The integer arg is converted to signed decimal (d), 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.

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.

The character `arg` is printed.

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:

```c
printf("%s, %s %d, %d:%.2d",
weekday, month, day, hour, min);
```

To print $\pi$ to 5 decimal places:

```c
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 $\infty$ and "not a number" (NaN: a symbolic entity encoded in floating point format) to support the IEEE P754 standard.

LEVEL

Level 1.
NAME
putc, putchar, fprintf, putw — put character or word on a stream

SYNOPSIS
#include <stdio.h>
int putc(c, stream)
int c;
FILE *stream;
int putchar(c)
int c;
int fprintf(c, stream)
int c;
FILE *stream;
int putw(w, stream)
int w;
FILE *stream;

DESCRIPTION
The function putc writes the character c onto the output stream at
the position where the file-pointer, if defined, is pointing.

The function putchar(c) is defined as follows:

putc(c, stdout)

Both putc and putchar are macros.

The function fprintf behaves like putc, but is a function rather than a
macro. The function fprintf runs more slowly than putc but it takes less
space per invocation and its name can be passed as an argument to a func­
tion.

The function putw writes the word (i.e., integer) w to the output
stream (where the file-pointer, if defined, is pointing). The size of a word
is the size of an integer and varies from machine to machine. The function
putw neither assumes nor causes special alignment in the file.

RETURN VALUE
On success, putc, fprintf, and putchar each return the value they
have written. On failure, they return the constant EOF. This will occur if
the file stream is not open for writing or if the output file cannot be
grown. The function putw returns non-zero when an error has occurred;
otherwise the function returns 0.
APPLICATION USAGE

Because it is implemented as a macro, _putc_ incorrectly treats the argument stream when it has side-effects. In particular, the following call may not work as expected:

```
putc(c, *f++);
```

The function _fputc_ should be used instead.

Because of possible differences in word length and byte ordering, files written using the function _putw_ are machine-dependent, and may not be read on a different processor using the function _getw_ [see GETC(BA_LIB)].

SEE ALSO

FCLOSE(BA_OS), FERROR(BA_OS), FOPEN(BA_OS), FREAD(BA_OS), PRINTF(BA_LIB), PUTS(BA_LIB), SETBUF(BA_LIB).

LEVEL

Level 1.
PUTENV(BA_LIB)

NAME
putenv — change or add value to environment

SYNOPSIS
int putenv(string)
char *string;

DESCRIPTION
The argument string points to a string of the following form:

name = value

The function putenv makes the value of the environment variable name equal to value by altering an existing variable or creating a new one. In either case, the string pointed to by string becomes part of the environment, so altering the string will change the environment. The space used by string is no longer used once a new string-defining name is passed to the function putenv.

RETURN VALUE
The function putenv returns non-zero if it was unable to obtain enough space for an expanded environment, otherwise zero.

APPLICATION USAGE
The function putenv was added to System V in System V Release 2.0.

The function putenv manipulates the environment pointed to by environ, and can be used in conjunction with getenv. However, envp, the third argument to main, is not changed [see EXEC(BA_OS)].

A potential error is to call the function putenv with a pointer to an automatic variable as the argument and to then exit the calling function while string is still part of the environment.

SEE ALSO
EXEC(BA_OS), MALLOC(BA_OS), GETENV(BA_LIB).

LEVEL
Level 1.
NAME
  puts, fputs — put a string on a stream

SYNOPSIS
  #include <stdio.h>
  int puts(s)
  char *s;

  int fputs(s, stream)
  char *s;
  FILE *stream;

DESCRIPTION
  The function puts writes the null-terminated string pointed to by s, fol-
  lowed by a new-line character, to the standard output stream stdout.

  The function fputs writes the null-terminated string pointed to by s to
  the named output stream.

  Neither function writes the terminating null character.

RETURN VALUE
  On success, both routines return the number of characters written.

  Both functions return EOF on error. This will happen if the routines try to
  write on a file that has not been opened for writing.

APPLICATION USAGE
  The function puts appends a new-line character while fputs does not.

SEE ALSO
  FERROR(BA_OS), FOPEN(BA_OS), FREAD(BA_OS), PRINTF(BA_LIB),
  PUTC(BA_LIB).

LEVEL
  Level 1.
QSORT(8A_LIB)

NAME
qsort — quicker sort

SYNOPSIS
void qsort(base, nel, width, compar)
char *base;
unsigned nel, width;
int (*compar)();

DESCRIPTION
The function qsort is a general-sorting algorithm. It sorts a table of data
in place.

The argument base points to the element at the base of the table.

The argument nel is the number of elements in the table.

The argument width is the size of an element in bytes.

The argument compar is the name of the user-supplied comparison func-
tion, which is called with two arguments that point to the elements being
compared. The comparison function must return an integer less than, equal
to or greater than zero, according as the first argument is to be considered is
less than, equal to or greater than the second.

APPLICATION USAGE
The pointer to the base the table should be of type pointer-to-element, and
cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may
be contained in the elements in addition to the values being compared.

The relative order in the output of two items which compare as equal is
unpredictable.

SEE ALSO
BSEARCH(8A_LIB), LSEARCH(8A_LIB), STRING(8A_LIB).

LEVEL
Level 1.
NAME
rand, srand — simple random-number generator

SYNOPSIS
int rand( )
void srand(seed)
unsigned int seed;

DESCRIPTION
The function rand uses a multiplicative congruential random-number generator with period $2^{32}$ that returns successive pseudo-random numbers in the range from 0 to 32767.

The function srand uses the argument seed as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to the function rand. If the function srand is then called with the same seed value, the sequence of pseudo-random numbers will be repeated. If the function rand is called before any calls to the function srand have been made, the same sequence will be generated as when the function srand is first called with a seed value of 1.

APPLICATION USAGE
The drand48(BA_LIB) routine provides a much more elaborate random-number generator.

The following functions define the semantics of the functions rand and srand.

```c
static unsigned long int next = 1;
int rand()
{
    next = next * 1103515245 + 12345;
    return ((unsigned int)(next/65536) % 32768);
}

void srand(unsigned int)(next/65536) % 32768);
{
    next = seed;
}
```

Specifying the semantics makes it possible to reproduce the behavior of programs that use pseudo-random sequences. This facilitates the testing of portable applications in different implementations.

SEE ALSO
DRAND48(BA_LIB).

LEVEL
Level 1.
NAME
regexp — regular-expression compile and match routines

SYNOPSIS
#define INIT declarations
#define GETC() getc code
#define PEEK() peekc code
#define UNGETC() ungetc code
#define RETURN(ptr) return code
#define ERROR(val) error code
#include <regexp.h>

char *compile(instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;
int eof;

int step(string, expbuf)
char *string, *endbuf;

advance(string, expbuf)
char *string, *expbuf;

extern char *loc1, *loc2, *locs;

DESCRIPTION
These functions are general-purpose regular-expression matching routines to be used in programs that perform regular-expression matching. These functions are defined by the <regexp.h> header file.

The functions step and advance do pattern matching given a character string and a compiled regular-expression as input.

The function compile takes as input a regular-expression as defined below and produces a compiled expression that can be used with step or advance.

A regular-expression, re, specifies a set of character strings. A member of this set of strings is said to be matched by the re. Some characters have special meaning when used in an re; other characters stand for themselves.

The regular-expressions available for use with the function regexp are constructed as follows:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>the character c where c is not a special character.</td>
</tr>
<tr>
<td>\c</td>
<td>the character c where c is any character, except a digit in the range 1–9.</td>
</tr>
<tr>
<td>^</td>
<td>the beginning of the line being compared.</td>
</tr>
<tr>
<td>$</td>
<td>the end of the line being compared.</td>
</tr>
</tbody>
</table>
any character in the set $s$, where $s$ is a sequence of characters and/or a range of characters, e.g., \[c-c\].

any character not in the set $s$, where $s$ is defined as above.

the occurrence of regular-expression $r$ followed by the occurrence of regular-expression $x$. (Concatenation)

any number of $m$ through $n$ successive occurrences of the regular-expression $r$. The regular-expression $r\{m\}$ matches exactly $m$ occurrences $r\{m,\}$ matches at least $m$ occurrences.

the regular-expression $r$. When \(n\) (where $n$ is a number greater than zero) appears in a constructed regular-expression, it stands for the regular-expression $x$ where $x$ is the $n^{th}$ regular-expression enclosed in \(\) and \(\) strings that appeared earlier in the constructed regular-expression. For example, \(r\(r\) x\(y\) z\2\) is the concatenation of regular-expressions $rxyz$.

Characters that have special meaning except when they appear within square brackets, \[,\] or are preceded by \(\) are: ., *, [ , \. Other special characters, such as $\$ have special meaning in more restricted contexts.

The character ^ at the beginning of an expression permits a successful match only immediately after a new-line, and the character $\$ at the end of an expression requires a trailing new-line.

Two characters have special meaning only when used within square brackets. The character - denotes a range, \[c-c\], unless it is just after the open bracket or before the closing bracket, \[-c\] or \[-c\] in which case it has no special meaning. When used within brackets, the character ^ has the meaning *complement* of if it immediately follows the open bracket, \[^{c}\], elsewhere between brackets, \[^{c}\], it stands for the ordinary character ^.

The special meaning of the \(\) operator can be escaped *only* by preceding it with another, e.g. \(\)\(\).
Programs must have the following five macros declared before the
#include <regexp.h> statement. These macros are used by the
compile routine. The macros GETC, PEEKC, and UNGETC operate on
the regular-expression given as input to compile.

GETC()  This macro returns the value of the next character in the
regular-expression pattern. Successive calls to GETC() should return successive characters of the regular-
expression.

PEEKC() This macro returns the next character in the regular-
expression. Immediately successive calls to PEEKC() should return the same character, which should also be
the next character returned by GETC().

UNGETC() This macro causes the argument c to be returned by the
next call to GETC() and PEEKC(). No more than
one character of pushback is ever needed and this charac-
ter is guaranteed to be the last character read by
GETC(). The value of the macro UNGETC(c) is
always ignored.

RETURN(ptr) This macro is used on normal exit of the compile rou-
tine. The value of the argument ptr is a pointer to the
character after the last character of the compiled regular-expression. This is useful to programs which have
memory allocation to manage.

ERROR(val) This macro is the abnormal return from the compile routine. The argument val is an error number [see
ERRORS below for meanings]. This call should never return.

The syntax of the compile routine is as follows:

    compile(instring, expbuf, endbuf, eof)

The first parameter instring is never used explicitly by the compile routine but is useful for programs that pass down different pointers to input
characters. It is sometimes used in the INIT declaration (see below). Pro-
grams which call functions to input characters or have characters in an exter-
nal array can pass down a value of ((char*)0) for this parameter.

The next parameter expbuf is a character pointer. It points to the place
where the compiled regular-expression will be placed.

The parameter endbuf is one more than the highest address where the
compiled regular-expression may be placed. If the compiled expression can-
ot fit in (endbuf-expbuf) bytes, a call to ERROR(50) is made.

The parameter eof is the character which marks the end of the regular-
expression. For example, rel.
Each program that includes the `<regexp.h>` header file must have a `#define` statement for INIT. It is used for dependent declarations and initializations. Most often it is used to set a register variable to point to the beginning of the regular-expression so that this register variable can be used in the declarations for `GETC()`, `PEEK( )`, and `UNGETC( )`. Otherwise it can be used to declare external variables that might be used by `GETC( )`, `PEEK( )` and `UNGETC( )`. See EXAMPLES below.

The first parameter to the `step` function is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter, `expbuf`, is the compiled regular-expression which was obtained by a call to the function `compile`.

The function `step` returns non-zero if some sub-string of `string` matches the regular-expression in `expbuf` and zero if there is no match. If there is a match, two external character pointers are set as a side effect to the call to `step`. The variable `loc1` points to the first character that matched the regular-expression; the variable `loc2` points to the character after the last character that matches the regular-expression. Thus if the regular-expression matches the entire input string, `loc1` will point to the first character of `string` and `loc2` will point to the null at the end of `string`.

The function `advance` returns non-zero if the initial substring of `string` matches the regular-expression in `expbuf`. If there is a match an external character pointer, `loc2`, is set as a side effect. The variable `loc2` points to the next character in `string` after the last character that matched.

When `advance` encounters a `*` or `{` `}` sequence in the regular-expression, it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular-expression. As long as there is no match, `advance` will back up along the string until it finds a match or reaches the point in the string that initially matched the `*` or `{` `}`. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer `loc2` is equal to the point in the string at some time during the backing up process, `advance` will break out of the loop that backs up and will return zero.

The external variables `circf`, `sed`, and `nbra` are reserved.
RETURN VALUE

The function compile uses the macro RETURN on success and the macro ERROR on failure, see above. The functions step and advance return non-zero on a successful match and zero if there is no match.

ERRORS

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>range endpoint too large.</td>
</tr>
<tr>
<td>16</td>
<td>bad number.</td>
</tr>
<tr>
<td>25</td>
<td>\digit out of range.</td>
</tr>
<tr>
<td>36</td>
<td>illegal or missing delimiter.</td>
</tr>
<tr>
<td>41</td>
<td>no remembered search string.</td>
</tr>
<tr>
<td>42</td>
<td>( ) imbalance.</td>
</tr>
<tr>
<td>43</td>
<td>too many (.</td>
</tr>
<tr>
<td>44</td>
<td>more than 2 numbers given in { }.</td>
</tr>
<tr>
<td>45</td>
<td>} expected after .</td>
</tr>
<tr>
<td>46</td>
<td>first number exceeds second in { }.</td>
</tr>
<tr>
<td>49</td>
<td>[ ] imbalance.</td>
</tr>
<tr>
<td>50</td>
<td>regular-expression overflow.</td>
</tr>
</tbody>
</table>

EXAMPLES

The following is an example of how the regular-expression macros and calls might be defined by an application program:

```c
#define INIT register char *sp = instring;
#define GETC() (*sp++)
#define PEEKC() (*sp)
#define UNGETC(c) (--sp)
#define RETURN(C) return;
#define ERROR(C) regerr()
#include <regexp.h>

... (void) compile(*argv, expbuf, &expbuf[ESIZE], '\0');
... if (step(linebuf, expbuf)) succeed();
```

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.
- `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 whitespace 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 \textit{first}–\textit{last}, thus \{0123456789\} may be expressed \([0-9]\). Using this convention, \textit{first} must be lexically less than or equal to \textit{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 \texttt{\textbackslash 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 \texttt{d}, \texttt{u}, \texttt{o}, and \texttt{x} may be preceded by \texttt{1} or \texttt{h} to indicate that a pointer to \texttt{long} or to \texttt{short} rather than to \texttt{int} is in the argument list. Similarly, the conversion characters \texttt{e}, \texttt{f}, and \texttt{g} may be preceded by \texttt{1} to indicate that a pointer to \texttt{double} rather than to \texttt{float} is in the argument list. The \texttt{1} or \texttt{h} modifier is ignored for other conversion characters.

The \texttt{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.

\textbf{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, \texttt{EOF} is returned.

\textbf{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.
**SCANF**(BA_LIB)

**EXAMPLE**

The call to the function `scanf`:

```c
int i, n; float x; char name[50];
```

```c
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`:

```c
int i; float x; char name[50];
```

```c
(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
setbuf, setvbuf — assign buffering to a stream

SYNOPSIS
#include <stdio.h>

void setbuf(stream, buf)
FILE *stream;
char *buf;

int setvbuf(stream, buf, type, size)
FILE *stream;
char *buf;
int type, size;

DESCRIPTION
The function setbuf may be used after a stream has been opened but
before it is read or written. It causes the array pointed to by buf to be
used instead of an automatically allocated buffer. If buf is the NULL
pointer input/output will be completely unbuffered.

A constant BUFSIZ, defined by the <stdio.h> header file, tells how
big an array is needed:

char buf[BUFSIZ];

The function setvbuf may be used after stream has been opened but
before it is read or written. The value of type determines how stream
will be buffered. Legal values for type, defined by the <stdio.h>
header file, are:

_IOFBF causes input/output to be fully buffered.
_IOLBF causes output to be line buffered; the buffer will be flushed when
a new-line is written, the buffer is full, or input is requested.
_IONBF causes input/output to be completely unbuffered.

If buf is not the NULL pointer, the array it points to will be used for
buffering instead of an automatically allocated buffer. The value of size
specifies the size of the buffer to be used. The constant BUFSIZ in the
<stdio.h> header file is suggested as a good buffer size. If input/output
is unbuffered, buf and size are ignored.

By default, output to a terminal is line buffered and all other input/output is
fully buffered, except the standard error stream stderr, which is normally
not buffered.

RETURN VALUE
If an illegal value for type or size is provided, setvbuf returns a
non-zero value. Otherwise, the value returned will be zero.
APPLICATION USAGE

The function setvbuf was added to System V in System V Release 2.0.

A common source of error is allocating buffer space as an automatic variable in a code block, and then failing to close the stream in the same block.

SEE ALSO

FOPEN(BA_OS), MALLOC(BA_OS), GETC(BA_LIB), PUTC(BA_LIB).

LEVEL

Level 1.
NAME
setjmp, longjmp — non-local goto

SYNOPSIS
#include <setjmp.h>

int setjmp(env)
jmp_buf env;
void longjmp(env, val)
jmp_buf env;
int val;

DESCRIPTION
These functions are useful for dealing with errors and interrupts encountered
in a low-level subroutine of a program.

The function setjmp saves its stack environment in env (whose type,
jmp_buf, is defined by the <setjmp.h> header file) for later use by
the function longjmp. The function setjmp returns the value 0.

The function longjmp restores the environment saved by the last call to
the function setjmp with the corresponding argument env.

After the function longjmp is completed, program execution continues as
if the corresponding call to the function setjmp (the caller of which must
not itself have returned in the interim) had just returned the value val.
All accessible data have values as of the time the function longjmp was
called.

RETURN VALUE
When the function setjmp has been called by the calling-process, it
returns 0.

The function longjmp does not return from where it was called, but
rather, program execution continues as if the previous call to the function
setjmp returned with a return value of val. That is, when the function
setjmp returns as a result of the function longjmp being called, the
function setjmp returns val. However, the function longjmp cannot
cause the function setjmp to return the value 0. If the function
longjmp is invoked with a val of 0, the function setjmp will return
1.

APPLICATION USAGE
If the function longjmp is called even though the argument env was
never primed by a call to the function setjmp, or when the last such call
was in a function which has since returned, the behavior is undefined.

If the call to the function longjmp is in a different function from the
corresponding call to the function setjmp, register variables may have
unpredictable values.
SETJMP(BA_LIB)

SEE ALSO
   SIGNAL(BA_OS).

LEVEL
   Level 1.
NAME
sinh, cosh, tanh — hyperbolic functions

SYNOPSIS
#include <math.h>
double sinh(x)
double x;
double cosh(x)
double x;
double tanh(x)
double x;

DESCRIPTION
The functions sinh, cosh, and tanh return, respectively, the hyperbolic sine, cosine and tangent of their argument.

RETURN VALUE
The functions sinh and cosh return HUGE, and sinh may return -HUGE for negative x, when the correct value would overflow and set errno to ERANGE.

APPLICATION USAGE
These error-handling procedures may be changed with the MATHERR(BA_LIB) routine.

SEE ALSO
MATHERR(BA_LIB).

FUTURE DIRECTIONS
A macro HUGE_VAL will be defined by the <math.h> header file. This macro will call a function which will either return +∞ on a system supporting the IEEE P754 standard or +{MAXDOUBLE} on a system that does not support the IEEE P754 standard.

The functions sinh and cosh will return HUGE_VAL (sinh will return -HUGE_VAL for negative x) when the correct value overflows.

LEVEL
Level 1.
SSIGNAL(BA_LIB)

NAME
ssignal, gsignal — software signals

SYNOPSIS
#include <signal.h>

int (*ssignal(sig, action))( )
int sig,(*action)();

int gsignal(sig)
int sig;

DESCRIPTION
The functions ssignal and gsignal implement a software facility
similar to the SIGNAL(BA_OS) routine. This facility is made available to pro-
grams for their own purposes.

Software signals available to programs are listed in SIGNAL(BA_OS).

A call to the function ssignal associates a procedure, action, with
the software signal sig; the software signal, sig, is raised by a call to the
function gsignal. Raising a software signal causes the action established
for that signal to be taken.

The first argument, sig, to the function ssignal, is a signal number in
the range 1–15 for which an action is to be established. The second argu-
ment, action, defines the action; it is either the name of a (user-defined)
function action or one of the manifest constants SIG_DFL (default) or
SIG_IGN (ignore). The function ssignal returns the action previously
established for that signal type; if no action has been established or the signal
is illegal, the function ssignal returns SIG_DFL.

The function gsignal raises the signal identified by its argument, sig:

If the function action has been established for the argument
sig, then that action is reset to SIG_DFL and the function
action is entered with argument sig. The function gsignal
returns the value returned to it by the function action.

If the action for the argument sig is SIG_IGN, the function
gsignal returns the value 1 and takes no other action.

If the action for the argument sig is SIG_DFL, the function
gsignal returns the value 0 and takes no other action.

If the argument sig has an illegal value or no action was ever
specified for the argument sig, the function gsignal returns the
value 0 and takes no other action.
SEE ALSO
  SIGNAL(BA_OS).

LEVEL
  Level 2, December 1, 1985
NAME
strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr,
strpbrk, strspn, strcspn, strtok — string operations

SYNOPSIS
#include <string.h>
char *strcat(s1, s2)
char *s1, *s2;
char *strncat(s1, s2, n)
char *s1, *s2;
int n;
int strcmp(s1, s2)
char *s1, *s2;
int strncmp(s1, s2, n)
char *s1, *s2;
int n;
char *strcpy(s1, s2)
char *s1, *s2;
char *strncpy(s1, *s2;
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 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 functions `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 `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 argument `n` to `strncat`, `strncmp` and `strncpy` and the type of value returned by `strlen` will be declared through the `typedef` facility in a header file as `size_t`.

LEVEL
Level 1.
NAME
strtod, atof — convert string to double-precision number

SYNOPSIS

double strtod(str, ptr)
char *str, **ptr;
double atof(str)
char *str;

DESCRIPTION
The function _strtod_ returns as a double-precision floating-point number the value represented by the character string pointed to by str. The string is scanned up to the first unrecognized character.

The function _strtod_ recognizes an optional string of white-space characters [as defined by isspace in _CTYPE(BA_LIB)_], then an optional sign, then a string of digits optionally containing a decimal point, then an optional e or E followed by an optional sign, followed by an integer.

If the value of _ptr_ is not ((char **)0), a pointer to the character terminating the scan is returned in the location pointed to by _ptr_. If no number can be formed, *ptr is set to str, and 0 is returned.

The function call _atof(str)_ is equivalent to:

```
strtod(str, (char **)0)
```

RETURN VALUE
If the correct value would cause overflow, ±HUGE is returned (according to the sign of the value) and _errno_ is set to ERANGE.

If the correct value would cause underflow, zero is returned and _errno_ is set to ERANGE.

APPLICATION USAGE
The function _strtod_ was added to System V in System V Release 2.0.

SEE ALSO
CTYPE(BA_LIB), SCNF(BA_LIB), STRTOL(BA_LIB).

FUTURE DIRECTIONS
A macro _HUGE_VAL_ will be defined by the <math.h> header file. This macro will call a function which will either return +∞ on a system that supports the IEEE P754 standard or +{MAXDOUBLE} on a system that does not support the IEEE P754 standard.

If the correct value overflows, ±HUGE_VAL will be returned (according to the sign of the value).

LEVEL
Level 1.
STRTOL(BA_LIB)

NAME
strtol, atol, atoi — convert string to integer

SYNOPSIS

long strtol(str, ptr, base)
char *str, **ptr;
int base;

long atol(str)
char *str;

int atoi(str)
char *str;

DESCRIPTION
The function `strtol` returns as a long integer the value represented by the character string pointed to by `str`. The string is scanned up to the first character inconsistent with the base. Leading white-space characters [as defined by `isspace` in `CTYPE(BA_LIB)`] are ignored.

If the value of `ptr` is not `((char **)0)`, a pointer to the character terminating the scan is returned in the location pointed to by `ptr`. If no integer can be formed, that location is set to `str` and zero is returned.

If `base` is positive (and not greater than 36), it is used as the base for conversion. After an optional leading sign, leading zeros are ignored and `0x` or `0X` is ignored if `base` is 16.

If `base` is zero, the string itself determines the base in the following way: After an optional leading sign a leading zero indicates octal conversion and a leading `0x` or `0X` hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from `long` to `int` can, of course, take place upon assignment or by an explicit cast.

The function call `atol(str)` is equivalent to:

`strtol(str, (char **)0, 10)`

The function call `atoi(str)` is equivalent to:

`(int)strtol(str, (char **)0, 10)`

RETURN VALUE
If the argument `ptr` is a null-pointer, the function `strtol` will return the value of the string `str` as a long integer.

If the argument `ptr` is not `NULL`, the function `strtol` will return the value of the string `str` as a long integer, and a pointer to the character terminating the scan will be returned in the location pointed to by `ptr`.

If no integer can be formed, that location is set to the argument `str` and the function `strtol` returns 0.
APPLICATION USAGE
Overflow conditions are ignored.

SEE ALSO
CTYPE(BA_LIB), SCANF(BA_LIB), STRTOD(BA_LIB).

FUTURE DIRECTIONS
Error handling will be added to the function strtol.

LEVEL
Level 1.
NAME
swab — swap bytes

SYNOPSIS
void swab(from, to, nbytes)
char *from, *to;
int nbytes;

DESCRIPTION
The function swab copies nbytes bytes pointed to by from to the array pointed to by to, exchanging adjacent even and odd bytes. It is useful for carrying binary data between machines with different low-order/high-order byte arrangements.

The argument nbytes should be even and non-negative. If the argument nbytes is odd and positive, the function swab uses nbytes-1 instead. If the argument nbytes is negative, the function swab does nothing.

LEVEL
Level 1.
NAME
tmpfile — create a temporary file

SYNOPSIS
#include <stdio.h>
FILE *tmpfile();

DESCRIPTION
The function tmpfile creates a temporary file using a name generated by
the TMPNAM(BA_LIB) library routine, and returns a corresponding pointer to
the FILE structure associated with the stream [see stdio-stream in Chapter
4 — Definitions]. The temporary file will automatically be deleted when the
process that opened it terminates or the temporary file is closed. The tem­
porary file is opened for update (w+) [see FOPEN(BA_OS)].

RETURN VALUE
If the temporary file cannot be opened, an error message is written and a
NULL pointer is returned.

SEE ALSO
CREAT(BA_OS), UNLINK(BA_OS), FOPEN(BA_OS), MKTEMP(BA_LIB),
TMPNAM(BA_LIB).

LEVEL
Level 1.
NAME
tmpnam, tempnam — create a name for a temporary file

SYNOPSIS
#include <stdio.h>
char *tmpnam(s)
char *s;
char *tempnam(dir, pfx)
char *dir, *pfx;

DESCRIPTION
These functions generate file-names that can safely be used for a temporary file.

The function tmpnam always generates a file-name using the path-prefix defined by the <stdio.h> header file as _tmpdir. If the argument s is NULL, the function tmpnam leaves its result in an internal static area and returns a pointer to that area. The next call to the function tmpnam will destroy the contents of the area. If the argument s is not NULL, it is assumed to be the address of an array of at least L_tmpnam bytes, where L_tmpnam is a constant defined by the <stdio.h> header file; the function tmpnam places its result in that array and returns s.

The function tempnam allows the user to control the choice of a directory. If defined in the user’s environment, the value of the environmental variable TMPDIR is used as the name of the desired temporary file directory. The argument dir points to the name of the directory in which the file is to be created. If the argument dir is NULL or points to a string that is not a name for an appropriate directory, the path-prefix defined by the <stdio.h> header file as _tmpdir is used. If that directory is not accessible, the directory /tmp will be used as a last resort.

The function tempnam uses the MALLOC(BA_OS) routine to get space for the constructed file-name, and returns a pointer to this area. Thus, any pointer value returned from the function tempnam may serve as an argument to the function free defined in MALLOC(BA_OS). If the function tempnam cannot return the expected result for any reason, for example, the MALLOC(BA_OS) routine failed or none of the above-mentioned attempts to find an appropriate directory was successful, a NULL pointer will be returned.

APPLICATION USAGE
Many applications prefer their temporary-files to have certain favorite initial letter sequences in their names. Use the pfx argument for this. This argument may be NULL or point to a string of up to five characters to be used as the first few characters of the temporary-file name.
The functions `tmpnam` and `tempnam` generate a different file-name each time they are called.

Files created using these functions and either the FOPEN(BA_OS) routine or the CREAT(BA_OS) routine are temporary only in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use the UNLINK(BA_OS) routine to remove the file when its use is ended.

If called more than (TMP_MAX) times in a single process, these functions will start recycling previously used names.

Between the time a file-name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or mktemp, and the file-names are chosen so as to render duplication by other means unlikely.

SEE ALSO
CREAT(BA_OS), UNLINK(BA_OS), FOPEN(BA_OS), MALLOC(BA_OS), MKTEMP(BA_LIB), TMPFILE(BA_LIB).

LEVEL
Level 1.
NAME
sin, cos, tan, asin, acos, atan, atan2 — trigonometric functions

SYNOPSIS
#include <math.h>

double sin(x)
double x;
double cos(x)
double x;
double tan(x)
double x;
double asin(x)
double x;
double acos(x)
double x;
double atan(x)
double x;
double atan2(y, x)
double y, x;

DESCRIPTION
The functions sin, cos and tan return respectively the sine, cosine and
tangent of their argument, x, measured in radians.

The function asin returns the arcsine of the argument x in the range
-\pi/2 to \pi/2.

The function acos returns the arccosine of the argument x in the range
0 to \pi.

The function atan returns the arctangent of the argument x in the range
-\pi/2 to \pi/2.

The function atan2 returns the arctangent of y/x in the range -\pi to \pi,
using the signs of both arguments to determine the quadrant of the return
value.

RETURN VALUE
Both sin and cos lose accuracy when their argument is far from zero.
For arguments sufficiently large, these functions return zero when there
would otherwise be a complete loss of significance. In this case a message
indicating TLOSS error is printed on the standard error output [see
MATHERR(3)]. For less extreme arguments causing partial loss of
significance, a PLOSS error is generated but no message is printed. In both
cases, errno is set to ERANGE.
If the magnitude of the argument of `asin` or `acos` is greater than one, or if both arguments of `atan2` are zero, zero is returned and `errno` is set to `EDOM`. In addition, a message indicating `DOMAIN` error is printed on the standard error output.

**APPLICATION USAGE**
These error-handling procedures may be changed with the `MATHERR(BA_LIB)` routine.

**SEE ALSO**
`MATHERR(BA_LIB)`.

**LEVEL**
Level 1.
TSEARCH(BA_LIB)

NAME

tsearch, tfind, tdelete, twalk — manage binary search trees

SYNOPSIS

#include <search.h>

char *tsearch(key, rootp, compar)
char *key;
char **rootp;
int (*compar)();

char *tfind(key, rootp, compar)
char *key;
char **rootp;
int (*compar)();

char *tdelete(key, rootp, compar)
char *key;
char **rootp;
int (*compar)();

void twalk(root, action)
char *root;
void(*action)();

DESCRIPTION

The functions tsearch, tfind, tdelete, and twalk manipulate binary search trees. All comparisons are done with a user-supplied function, compar. The comparison function is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The function tsearch is used to build and access the tree. The value of key is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to *key (the value pointed to by key), a pointer to this found datum is returned. Otherwise, *key is inserted, and a pointer to it is returned. Only pointers are copied, so the calling routine must store the data. The value of rootp points to a variable that points to the root of the tree. A NULL value for the variable pointed to by rootp denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like tsearch, tfind will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, tfind will return a NULL pointer. The arguments for tfind are the same as for tsearch.

The function tdelete deletes a node from a binary search tree. The arguments are the same as for tsearch. The variable pointed to by rootp will be changed if the deleted node was the root of the tree.
The function `twalk` traverses a binary search tree. The value of `root` is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) The value of `action` is the name of a user-defined routine to be invoked at each node. This routine is, in turn, called with three arguments.

The first argument is the address of the node being visited.

The second argument is a value from an enumeration data type, `VISIT` defined by the `<search.h>` header file. The values `preorder`, `postorder`, `endorder`, indicate whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or the value `leaf` indicates that the node is a leaf.

The third argument is an integer that identifies the level of the node in the tree, with the root being level zero.

**RETURN VALUE**

A NULL pointer is returned by `tsearch` if there is not enough space available to create a new node.

A NULL pointer is returned by `tsearch`, `tfind` and `tdelete` if `rootp` is NULL on entry.

If the datum is found, both `tsearch` and `tfind` return a pointer to it. If not, `tfind` returns NULL, and `tsearch` returns a pointer to the inserted item. The function `tdelete` returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

**APPLICATION USAGE**

The function `tfind` was added to System V in System V Release 2.0.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

The `root` argument to `twalk` is one level of indirection less than the `rootp` arguments to `tsearch` and `tdelete`.

There are two nomenclatures used to refer to the order in which tree nodes are visited. The function `tsearch` uses preorder, postorder and endorder to respectively refer to visiting a node before any of its children, after its left child and before its right, and after both its children. The alternate nomenclature uses preorder, inorder and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder.

If the calling function alters the pointer to the root, results are unpredictable.
EXAMPLE

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

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

struct node {
    char *string;
    int length;
};
char string_space[10000]; /* space to store strings */
struct node nodes[500]; /* nodes to store */
struct node *root = NULL; /* this points to the root */

main( )
{
    char *strptr = string_space;
    struct node *nodeptr = nodes;
    void print_node(), twalk();
    int i = 0, node_compare();
    while (gets(strptr) != NULL && i++ < sizeof(nodes [0])) {
        /* set node */
        nodeptr->string = strptr;
        nodeptr->length = strlen(strptr);
        /* put node into the tree */
        (void) tsearch((char *)nodeptr, &root, node_compare);
        /* adjust pointers, to not overwrite tree */
        strptr += nodeptr->length + 1;
        nodeptr++;
    }
    twalk(root, print_node);
}
/* This routine compares two nodes, based on an */
/* alphabetical ordering of the string field. */
int node_compare(node1, node2)
    struct node *node1, *node2;
{
    return strcmp(node1->string, node2->string);
}
/* This routine prints out a node, the */
/* first time twalk encounters it. */
void print_node(node, order, level)
    struct node **node;
    VISIT order;
    int level;
{
    if (order == preorder || order == leaf) {
        (void) printf("string = %20s, length = %d\n", 
            (*node)->string, (*node)->length);
    }
}
```
SEE ALSO
BSEARCH(BA_LIB), HSEARCH(BA_LIB), LSEARCH(BA_LIB).

LEVEL
Level 1.
TTYNAME(BA_LIB)

NAME

ttyname, isatty — find name of a terminal

SYNOPSIS

char *ttyname(fildes)
int fildes;

int isatty(fildes)
int fildes;

DESCRIPTION

The function ttyname returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor fildes.

The function isatty returns 1 if the argument fildes is associated with a terminal device, 0 otherwise.

RETURN VALUE

The function ttyname returns a null-pointer if the argument fildes does not describe a terminal device.

APPLICATION USAGE

The return value points to static data whose content is overwritten by each call.

LEVEL

Level 1.
NAME
  ungetc — push character back into input stream

SYNOPSIS
  #include <stdio.h>
  int ungetc(c, stream)
  int c;
  FILE *stream;

DESCRIPTION
  The function ungetc inserts the character c into the buffer associated
  with an input stream. That character, c, will be returned by the next
  call to the GETC(BA_LIB) routine on that stream. The function ungetc
  returns c, and leaves the file corresponding to stream unchanged.

  One character of pushback is guaranteed, provided something has already
  been read from the stream and the stream is actually buffered.

  If the argument c equals EOF, the function ungetc does nothing to the
  buffer and returns EOF.

  The FSEEK(BA_OS) routine erases all memory of inserted characters.

RETURN VALUE
  If successful, the function ungetc returns c; the function ungetc
  returns EOF if it cannot insert the character.

SEE ALSO
  FSEEK(BA_OS), GETC(BA_LIB), SETBUF(BA_LIB).

LEVEL
  Level 1.
VPRTF(BA_LI B)

NAME

vprintf, vfprintf, vsprintf — print formatted output of a varargs argument list

SYNOPSIS

#include <stdio.h>
#include <varargs.h>

int vprintf(format, ap)
char *format;
va_list ap;

int vfprintf(stream, format, ap)
FILE *stream;
char *format;
va_list ap;

int vsprintf(s, format, ap)
char *s, *format;
va_list ap;

DESCRIPTION

The functions vprintf, vfprintf, and vsprintf are the same as printf, fprintf, and sprintf respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by the <varargs.h> header file.

The <varargs.h> header file defines the type va_list and a set of macros for advancing through a list of arguments whose number and types may vary. The argument ap to the vprintf family of library routines is of type va_list. This argument is used with the <varargs.h> header file macros va_start, va_arg and va_end. The EXAMPLE section below shows their use with vprintf.

The macro va_al ist is used as the parameter list in a function definition as in the function called error in the example below. The macro va_dcl is the declaration for va_al ist and should not be followed by a semicolon. The macro va_start(ap), where ap is of type va_list, must be called before any attempt to traverse and access the list of arguments. Calls to va_arg(ap, atype) traverse the argument list. Each execution of va_arg expands to an expression with the value of the next argument in the list ap. The argument atype is the type that the returned argument is expected to be. The va_end(ap) macro must be executed when all desired arguments have been accessed. (The argument list in ap can be traversed again if va_start is called again after va_end.) In the example below, va_arg is executed first to return the function_name passed to error and it is called again to retrieve the format passed to error. The remaining error arguments, arg1, arg2, ..., are given to vfprintf in the argument ap.
APPLICATION USAGE

The functions vprintf, vfprintf and vsprintf were added to System V in System V Release 2.0.

EXAMPLE

The following demonstrates how vfprintf could be used to write an error routine:

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

/*
 * error should be called like
 * error(function_name, format, arg1, arg2...);
 */
void error(va_alist)
va_dcl

{ va_list ap;
  char *fmt;
  va_start(ap);
  /* print out name of function causing error */
  (void) fprintf(stderr, "ERR in %s:
  fmt = va_arg(ap, char *);
  /* print out remainder of message */
  (void) vfprintf(stderr, fmt, ap);
  va_end(ap);
  (void) abort();
}
```

SEE ALSO

PRINTF(BA_LIB).

LEVEL

Level 1.
Part III

Kernel Extension Definition
Chapter 8
Introduction

While the Base System is intended to support a run-time environment for executable applications, the Kernel Extension provides additional operating system services that will not be required by many application-programs but which are needed for some environments.

The Kernel Extension provides operating system services to support process accounting tools, software development tools, and applications or tools that require more sophisticated inter-process communication than is provided by the Base System.

Definitions for the Kernel Extension are given in the next chapter, Chapter 8 — Definitions. Chapter 9 — Environment describes the Kernel Extension Environment including additional behavior of Base System components when the Kernel Extension is present on a system [see EFFECTS(KE_ENV)]. Chapter 10 — OS Service Routines has the component definitions of the operating system services in the Kernel Extension.

The following operating system services constitute the System V Kernel Extension. An application-program that uses any of these components would require the target run-time environment to support the Kernel Extension in addition to the Base System.

TABLE 8-1. Kernel Extension: OS Service Routines

<table>
<thead>
<tr>
<th>acct</th>
<th>ACCT(KE_OS)</th>
<th>ptrace</th>
<th>PTRACE(KE_OS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>chroot</td>
<td>CHROOT(KE_OS)</td>
<td>semctl</td>
<td>SEMCTL(KE_OS)</td>
</tr>
<tr>
<td>msgctl</td>
<td>MSGCTL(KE_OS)</td>
<td>semget</td>
<td>SEMGET(KE_OS)</td>
</tr>
<tr>
<td>msgget</td>
<td>MSGGET(KE_OS)</td>
<td>semop</td>
<td>SEMOP(KE_OS)</td>
</tr>
<tr>
<td>msgrcv</td>
<td>MSGOP(KE_OS)</td>
<td>shmctl#</td>
<td>SHMCTL(KE_OS)</td>
</tr>
<tr>
<td>msgsnd</td>
<td>MSGOP(KE_OS)</td>
<td>shmget#</td>
<td>SHMGET(KE_OS)</td>
</tr>
<tr>
<td>nice</td>
<td>NICE(KE_OS)</td>
<td>shm#</td>
<td>SHMOP(KE_OS)</td>
</tr>
<tr>
<td>plock</td>
<td>PLOCK(KE_OS)</td>
<td>shmdt#</td>
<td>SHMOP(KE_OS)</td>
</tr>
<tr>
<td>profil</td>
<td>PROFIL(KE_OS)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The run-time behavior of these routines, which is supported by the Kernel Extension, and the source-code interface to the routines are defined in Chapter 10 — OS Service Routines.

# Optional. These routines are hardware-dependent and will only appear on machines with the appropriate hardware.
Chapter 9
Definitions
ipc-permissions
The Kernel Extension includes three mechanisms for inter-process communication (ipc): messages, semaphores, and shared-memory. All of these use a common structure type, *ipc_perm*, to pass information used in determining permission to perform an ipc operation.

The *ipc_perm* structure is defined by the *<ipc.h>* header file and includes the following members:

- ushort *cuid*; /* creator user id */
- ushort *cgid*; /* creator group id */
- ushort *uid*; /* user id */
- ushort *gid*; /* group id */
- ushort *mode*; /* r/w permission */

The following symbolic constants are also defined by the *<ipc.h>* header file:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPC_CREATE</td>
<td>create entry if key does not exist</td>
</tr>
<tr>
<td>IPC_EXCL</td>
<td>fail if key exists</td>
</tr>
<tr>
<td>IPC_NOWAIT</td>
<td>error if request must wait</td>
</tr>
<tr>
<td>IPC_PRIVATE</td>
<td>private key</td>
</tr>
<tr>
<td>IPC_RMID</td>
<td>remove identifier</td>
</tr>
<tr>
<td>IPC_SET</td>
<td>set options</td>
</tr>
<tr>
<td>IPC_STAT</td>
<td>get options</td>
</tr>
</tbody>
</table>

message-queue-identifier
A message queue identifier *msqid* is a unique positive integer created by a call to the MSGGET(KE_OS) routine. Each *msqid* has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```c
struct ipc_perm msg_perm; /* operation perms */
ushort msg_qnum; /* no. of messages on q */
ushort msg_qbytes; /* max no. of bytes on q */
ushort msg_lspid; /* pid, last msgsnd call */
ushort msg_lrpid; /* pid, last msgrcv call */
time_t msg_stime; /* last msgsnd time */
time_t msg_rtime; /* last msgrcv time */
time_t msg_ctime; /* last change time */
```

/* time in secs since 00:00:00 GMT 1 Jan 70 */
msg_perm is an ipc_perm structure [see ipc-permissions] that specifies the message-operation permission.

msg_qnum is the number of messages currently on the queue.

msg_qbytes is the maximum number of bytes allowed on the queue.

msg_lspid is the process-ID of the last process that performed a msgsnd operation.

msg_lrpid is the process-ID of the last process that performed a msgrcv operation.

msg_stime is the time of the last msgsnd operation.

msg_rtime is the time of the last msgrcv operation.

msg_ctime is the time of the last msgctl operation that changed a member of the above structure.

**message-operation-permissions**

In the MSGOP(KE_OS) and MSGCTL(KE_OS) routines, the permission required for an operation is determined by the bit-pattern in msg_perm.mode, where the type of permission needed is interpreted as follows:

<table>
<thead>
<tr>
<th>Code</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>00400</td>
<td>Read by user</td>
</tr>
<tr>
<td>00200</td>
<td>Write by user</td>
</tr>
<tr>
<td>00040</td>
<td>Read by group</td>
</tr>
<tr>
<td>00020</td>
<td>Write by group</td>
</tr>
<tr>
<td>00004</td>
<td>Read by others</td>
</tr>
<tr>
<td>00002</td>
<td>Write by others</td>
</tr>
</tbody>
</table>

The Read and Write permissions on a msqid 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 msg_perm.cuid or msg_perm.uid in the data structure associated with msqid and the appropriate bit of the user portion (0600) of msg_perm.mode is set.
- The effective-user-ID of the process does not match msg_perm.cuid or msg_perm.uid, and the effective-group-ID of the process matches msg_perm.cgid or msg_perm.gid, and the appropriate bit of the group portion (0060) of msg_perm.mode is set.
- The effective-user-ID of the process does not match msg_perm.cuid or msg_perm.uid, and the effective-group-ID of the process does not match msg_perm.cgid or msg_perm.gid, and the appropriate bit of the other portion (0006) of msg_perm.mode is set.

Otherwise, the corresponding permissions are denied.
A semaphore-identifier semid is a unique positive integer created by a SEMGET(KE_OS) routine. Each semid has a set of semaphores and a data structure associated with it.

The data structure is semid_ds and contains the following members:

```c
struct ipc_perm sem_perm; /* operation perms */
ushort sem_nsems; /* count of sems in set */
time_t sem_otime; /* last operation time */
time_t sem_ctime; /* last change time */
    /* time in secs since */
    /* 00:00:00 GMT 1 Jan 70 */
```

sem_perm is an ipc_perm structure that specifies the semaphore-operation-permission [see ipc-permissions].

sem_nsems is a value that is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a sem_num. The value of sem_num runs sequentially from 0 to the value of sem_nsems-1.

sem_otime is the time of the last semop operation, and sem_ctime is the time of the last semctl operation that changed a member of the above structure.

A semaphore is a data structure containing the following members:

```c
ushort semval; /* semaphore value */
short sempid; /* pid of last operation */
ushort semncnt; /* no. awaiting semval > cval */
ushort semzcnt; /* no. awaiting semval = 0 */
```

semval is a non-negative integer.

sempid is equal to the process-ID of the last process that performed a semaphore operation on this semaphore.

semncnt is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become greater than its current value.

semzcnt is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.
**Semaphore-Operation-Permissions**

In the SEMOP(KE_OS) and SEMCTL(KE_OS) routines, the permission required for an operation is determined by the bit-pattern in `sem_perm.mode`, where the type of permission needed is interpreted as follows:

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>00400</td>
<td>Read by user</td>
</tr>
<tr>
<td>00200</td>
<td>Alter by user</td>
</tr>
<tr>
<td>00040</td>
<td>Read by group</td>
</tr>
<tr>
<td>00020</td>
<td>Alter by group</td>
</tr>
<tr>
<td>00004</td>
<td>Read by others</td>
</tr>
<tr>
<td>00002</td>
<td>Alter by others</td>
</tr>
</tbody>
</table>

The Read and Alter permissions on a `semid` 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 `sem_perm.cuid` or `sem_perm.uid` in the data structure associated with `semid` and the appropriate bit of the user portion (0600) of `sem_perm.mode` is set.
- The effective-user-ID of the process does not match `sem_perm.cuid` or `sem_perm.uid`, and the effective-group-ID of the process matches `sem_perm.cgid` or `sem_perm.gid`, and the appropriate bit of the group portion (0060) of `sem_perm.mode` is set.
- The effective-user-ID of the process does not match `sem_perm.cuid` or `sem_perm.uid`, and the effective-group-ID of the process does not match `sem_perm.cgid` or `sem_perm.gid`, and the appropriate bit of the other portion (0006) of `sem_perm.mode` is set.

Otherwise, the corresponding permissions are denied.

**Shared-Memory-Identifier**

A shared-memory-identifier `shmid` is a unique positive integer created by a SHMGET(KE_OS) routine. Each `shmid` has associated with it a segment of memory (referred to as a shared memory segment) and a data structure.

The data structure is referred to as `shmid_ds` and contains the following members:

```c
struct ipc_perm shm_perm; /* operation perms */
int shm_segsz; /* size of segment */
ushort shm_cpid; /* pid, creator */
ushort shm_lpid; /* pid, last operation */
short shm_nattch; /* no. of current attaches */
time_t shm_atime; /* last attach time */
time_t shm_dtime; /* last detach time */
time_t shm_ctime; /* last change time */
/* times in secs since */
/* 00:00:00 GMT 1 Jan 70 */
```
shm_perm is an ipc_perm structure that specifies the shared-memory-operation permission [see ipc-permissions].

shm_segsz specifies the size of the shared-memory-segment.

shm_cpid is the process-ID of the process that created the shared-memory-identifier.

shm_lpid is the process-ID of the last process that performed a SHMOP(KE_OS) routine.

shm_nattch is the number of processes that currently have this segment attached.

shm_atime is the time of the last shmat operation.

shm_dtime is the time of the last shmdt operation.

shm_ctime is the time of the last shmctl operation that changed one of the members of the above structure.

shared-memory-operation-permissions

In the SHMOP(KE_OS) and SHMCTL(KE_OS) routines, the permission required for an operation is determined by the bit-pattern in shm_perm.mode, where the type of permission needed is interpreted as follows:

- 00002  Write by others
- 00004  Read by group
- 00020  Write by group
- 00200  Write by user
- 00400  Read by user

The Read and Write permissions on a shmid 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 shm_perm.cuid or sem_perm.uid in the data structure associated with shmid and the appropriate bit of the user portion (0600) of shm_perm.mode is set.
- The effective-user-ID of the process does not match shm_perm.cuid or sem_perm.uid, and the effective-group-ID of the process matches shm_perm.cgid or sem_perm.gid, and the appropriate bit of the group portion (0060) of shm_perm.mode is set.
- The effective-user-ID of the process does not match shm_perm.cuid or sem_perm.uid, and the effective-group-ID of the process does not match shm_perm.cgid or sem_perm.gid, and the appropriate bit of the other portion (0006) of shm_perm.mode is set.

Otherwise, the corresponding permissions are denied.
Chapter 10
Environment
EFFECTS(KE_ENV)

NAME
effects — effects of the Kernel Extension on the Base System.

DESCRIPTION
Some of the Base System V operating system services are affected by the additional services in this extension. The effects are listed below for each routine:

EXEC(BA_OS)

The AFORK flag in the ac_flag field of the accounting record is turned off, and the ac_comm field is reset by executing an exec routine [see ACCT(KE_OS)].

Any process, data, or text locks are removed and not inherited by the new process [see PLOCK(KE_OS)].

Profiling is disabled for the new process [see PROFIL(KE_OS)].

The shared-memory-segments attached to the calling-process will not be attached to the new process [see SHMOP(KE_OS)].

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

- nice value [see NICE(KE_OS)];
- semadj values [see SEMOP(KE_OS)];
- trace flag [see request 0 in PTRACE(KE_OS)].

EXIT(BA_OS)

An accounting record is written on the accounting file if the system's accounting routine is enabled [see ACCT(KE_OS)].

If the process has a process-lock, text-lock, or data-lock, the lock is removed [see PLOCK(KE_OS)].

Each attached shared-memory-segment is detached and the value of shm_nattch in the data structure associated with its shared-memory-identifier is decremented by 1.

For each semaphore for which the calling-process has set a semadj value [see SEMOP(KE_OS)], that semadj value is added to the semval of the specified semaphore.
The *AFORK* flag is turned on when the function `fork` is executed.

The child-process inherits the following additional attributes from the parent-process:

- The `ac_comm` contents of the accounting record ([see `ACCT(KE_OS)`](#));
- nice value ([see `NICE(KE_OS)`](#));
- profiling on/off status ([see `PROFIL(KE_OS)`](#));
- all attached shared-memory-segments ([see `SHMOP(KE_OS)`](#)).

The child-process differs from the parent-process in the following additional ways:

- All `semadj` values are cleared ([see `SEMOP(KE_OS)`](#)).
- Process-locks, text-locks, and data-locks are not inherited by the child-process ([see `PLOCK(KE_OS)`](#)).

**LEVEL**

Level 1.
ERRNO(KE_ENV)

NAME
error — error codes and condition definitions

SYNOPSIS
#include <errno.h>
extern int errno;

DESCRIPTION
In addition to the values defined in the Base System for the external variable errno [see ERRNO(BA_ENV)], two additional error conditions are defined in the Kernel Extension:

ENO MSG  No message of desired type.
An attempt was made to receive a message of a type that does not exist on the specified message queue.

EIDRM  Identifier removed.
This error is returned to processes that resume execution because of the removal of an identifier [see MSGCTL(KE_OS), SEMCTL(KE_OS), and SHMCTL(KE_OS)].

LEVEL
Level 1.
Chapter 11
OS Service Routines
NAME
acct — enable or disable process accounting

SYNOPSIS
int acct(path)
    char *path;

DESCRIPTION
The function acct is used to enable or disable the system process accounting routine. If the routine is enabled, for each process that terminates, an accounting record will be written on an accounting file. Termination can be caused by one of two things: an exit call or a signal [see EXIT(BA_OS) and SIGNAL(BA_OS)]. The effective-user-ID of the calling-process must be super-user to use this function.

The variable path points to a path-name naming the accounting file. The format of an accounting file produced as a result of calling the acct function has records in the format defined by the structure acct in <sys/acct.h> which defines the following data-type:

```
comp_t /* floating point - 13-bit fraction, */
    /* 3-bit exponent */
```

and defines the following members in the structure acct:

```
char    ac_flag; /* accounting flag */
char    ac_stat; /* exit status */
ushort   ac_uid; /* accounting user-ID */
ushort   ac_gid; /* accounting group-ID */
dev_t    ac_tty; /* control typewriter */
time_t   ac_btime; /* beginning time */
comp_t   ac_utime; /* user-time in CLKTCKs */
comp_t   ac_stime; /* system-time in CLKTCKs */
comp_t   ac_etime; /* elapsed-time in CLKTCKs */
comp_t   ac_mem; /* memory usage */
comp_t   ac_io; /* chars transferred */
comp_t   ac_rw; /* blocks read or written */
char    ac_comm[8]; /* command name */
```

and defines the following symbolic names:

```
AFORK /* has executed fork, but no exec */
ASU    /* used super-user privileges */
ACCTF  /* record type: 00 = acct */
```

The AFORK flag is set in ac_flag when the FORK(BA_OS) routine is executed and reset when an EXEC(BA_OS) routine is executed. The ac_comm field is inherited from the parent process when a child process is created with the FORK(BA_OS) routine and is reset when the EXEC(BA_OS) routine is executed. The variable ac_mem is a cumulative record of memory usage and is incremented each time the system charges the process with a clock tick.
The accounting routine is enabled if `path` is non-zero and no errors occur during the call. It is disabled if `path` is 0 and no errors occur during the call.

**RETURN VALUE**
If successful, the function `acct` returns 0; otherwise, it returns -1 and `errno` will indicate the error.

**ERRORS**
The function `acct` will fail if one or more of the following are true:

- **EPERM** The effective user of the calling-process is not super-user.
- **EBUSY** An attempt is being made to enable accounting when it is already enabled.
- **ENOTDIR** A component of the path-prefix is not a directory.
- **ENOENT** One or more components of the accounting file path-name do not exist.
- **EACCES** The file named by `path` is not an ordinary file.
- **EROFS** The named file resides on a read-only file system.

**SEE ALSO**
`EXIT(BA_OS), SIGNAL(BA_OS)`.

**LEVEL**
Level 1.
CHROOT(KE_OS)

NAME
chroot — change root directory

SYNOPSIS
int chroot(path)
char *path;

DESCRIPTION
The function chroot causes the named directory to become the root directory, the starting point for path searches for path-names beginning with the character '/'. The user's working directory is unaffected by the function chroot.

The argument path points to a path-name naming a directory.

The effective-user-ID of the process must be super-user to change the root directory.

The .. entry in the root directory is interpreted to mean the root directory itself. Thus, .. cannot be used to access files outside the sub-tree rooted at the root-directory.

RETURN VALUE
If successful, the function chroot returns 0; otherwise, it returns −1 and errno will indicate the error.

ERRORS
The function chroot will fail and the root directory will remain unchanged if one or more of the following are true:

ENOTDIR Any component of the path-name is not a directory.
ENOENT The named directory does not exist.
EPERM The effective-user-ID is not super-user.

SEE ALSO
CHDIR(BA_OS).

LEVEL
Level 1.
NAME
msgctl — message-control-operations

SYNOPSIS
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;

DESCRIPTION
The function msgctl provides a variety of message-control-operations as specified by cmd. The following values for cmd and the message-control-operations they specify are available:

IPC_STAT
Place the current value of each member of the data structure associated with msqid into the structure pointed to by buf. The contents of this structure are defined in Chapter 9 — Definitions.

IPC_SET
Set the value of the following members of the data structure associated with msqid to the corresponding value found in the structure pointed to by buf:

msg_perm.uid
msg_perm.gid
msg_perm.mode /* only low 9-bits */
msg_qbytes

This cmd can only be executed by a process that has an effective-user-ID equal to either that of super-user or to the value of msg_perm.cuid or msg_perm.uid in the data structure associated with msqid. Only super-user can raise the value of msg_qbytes.

IPC_RMICD
Remove the message-queue-identifier specified by msqid from the system and destroy the message-queue and data structure associated with it. This cmd can only be executed by a process that has an effective-user-ID equal to either that of super-user or to the value of msg_perm.cuid or msg_perm.uid in the data structure associated with msqid.

RETURN VALUE
If successful, the function msgctl returns 0; otherwise, it returns -1 and errno will indicate the error.
MSGCTL(KE_OS)

ERRORS
The function msgctl will fail if one or more of the following are true:

EINVAL The value of msqid is not a valid message-queue-identifier; or
the value of cmd is not a valid command.

EACCES The argument cmd is equal to IPC_STAT and the calling-
process does not have read permission [see message-operation-
permissions in Chapter 9 — Definitions].

EPERM The argument cmd is equal to IPC_RMD or IPC_SET and
the effective-user-ID of the calling-process is not equal to that of
super-user and it is not equal to the value of msg_perm.cuid
or msg_perm.uid in the data structure associated with
msqid.

EPERM The argument cmd is equal to IPC_SET, an attempt is being
made to increase to the value of msg_qbytes, and the
effective-user-ID of the calling-process is not equal to that of
super-user.

SEE ALSO
MSGGET(KE_OS), MSGOP(KE_OS).

LEVEL
Level 1.
NAME
msgget — get message-queue

SYNOPSIS
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key, msgflg)
key_t key;
int msgflg;

DESCRIPTION
The function msgget returns the message-queue-identifier associated with
the argument key.

A message-queue-identifier and associated message-queue and data structure
[see Chapter 9 — Definitions] are created for the argument key if one of
the following are true:

if the argument key is equal to IPC_PRIVATE.

if the argument key does not already have a message-queue-
identifier associated with it, and (msgflg&IPC_CREAT) is true.

Upon creation, the data structure associated with the new message-queue-
identifier is initialized as follows:

msg_perm.cuid and msg_perm.uid are set equal to the
effective-user-ID of the calling-process;

msg_perm.cgid, and msg_perm.gid are set equal to the
effective-group-ID of the calling-process;

The low-order 9-bits of msg_perm.mode are set equal to the
low-order 9-bits of msgflg;

msg_qnum, msg_lspid, msg_lrpid, msg_stime, and
msg_rtime are set equal to 0;

msgctime is set equal to the current-time;

msg_qbytes is set equal to the system-limit.

RETURN VALUE
If successful, the function msgget returns a non-negative integer, namely a
message-queue-identifier; otherwise, it returns -1 and errno will indicate
the error.
MSGGET(KE_OS)

ERRORS
The function msgget will fail if one or more of the following are true:

EACCES A message-queue-identifier exists for the argument key, but operation permission [see Chapter 9 — Definitions] as specified by the low-order 9-bits of msgflg would not be granted.

ENOENT A message-queue-identifier does not exist for the argument key and (msgflg&IPC_CREAT) is "false".

ENOSPC A message-queue-identifier is to be created but the system-imposed limit on the maximum number of allowed message-queue-identifiers system-wide would be exceeded.

EEXIST A message-queue-identifier exists for the argument key but ((msgflg&IPC_CREAT)&&(msgflg&IPC_EXCL)) is "true".

SEE ALSO
MSGCTL(KE_OS), MSGOP(KE_OS).

LEVEL
Level 1.
NAME

msgop — message operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(msqid, msgp, msgsz, msgflg)
int msqid;
struct mymsg *msgp;
int msgsz, msgflg;

int msgrcv(msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct mymsg *msgp;
int msgsz;
long msgtyp;
int msgflg;

DESCRIPTION

The function msgsnd is used to send a message to the queue associated
with the message queue identifier specified by msqid.

The argument msgp points to a user-defined buffer that must contain first a
field of type long integer that will specify the type of the message, and then a
data portion that will hold the text of the message. The structure below is an
example of what this user-defined buffer might look like.

struct mymsg {
    long mtype; /* message type */
    char mtext[]; /* message text */
}

The structure member mtype is a positive integer that can be used by the
receiving process for message selection (see msgrcv below).

The structure member mtext is any text of length msgsz bytes. The
argument msgsz can range from 0 to a system-imposed maximum.

The argument msgflg specifies the action to be taken if one or more of
the following are true:

The number of bytes already on the queue is equal to
msg_qbytes [see Chapter 9 - Definitions].

The total number of messages on all queues system-wide is equal to
the system-imposed limit.
These actions are as follows:

If \((\text{msgflg} \& \text{IPC_NOWAIT})\) is "true", the message will not be sent and the calling-process will return immediately.

If \((\text{msgflg} \& \text{IPC_NOWAIT})\) is "false", the calling-process will suspend execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in which case the message is sent.

The message-queue-identifier \(\text{msqid}\) is removed from the system [see \text{MSGCTL(KE_OS)}]. When this occurs, \(\text{errno}\) is set equal to \(\text{EIDRM}\) and a value of \(-1\) is returned.

The calling-process receives a signal that is to be caught. In this case the message is not sent and the calling-process resumes execution in the manner prescribed in the \text{SIGNAL(BA_OS)} routine.

Upon successful completion, the following actions are taken with respect to the data structure associated with \(\text{msqid}\) [see \text{Chapter 9 - Definitions}].

\(\text{msg_qnum}\) is incremented by \(1\).

\(\text{msg_lspid}\) is set equal to the process-ID of the calling-process.

\(\text{msg_stime}\) is set equal to the current time.

The function \text{msgrcv} reads a message from the queue associated with the message queue identifier specified by \(\text{msqid}\) and places it in the user-defined buffer pointed to by \(\text{msgp}\). The buffer must contain a message type field followed by the area for the message text (see the structure \text{mymsg} above).

The structure member \(\text{mtype}\) is the received message's type as specified by the sending process.

The structure member \(\text{mtext}\) is the text of the message.

The argument \(\text{msgsz}\) specifies the size in bytes of \(\text{mtext}\). The received message is truncated to \(\text{msgsz}\) "bytes" if it is larger than \(\text{msgsz}\) and \((\text{msgflg} \& \text{MSG_NOERROR})\) is "true". The truncated part of the message is lost and no indication of the truncation is given to the calling-process.

The symbolic name \text{MSG_NOERROR} is defined by the \(<\text{sys/msg.h}>\) header file.
The argument `msgtyp` specifies the type of message requested as follows:

- If `msgtyp` is equal to 0, the first message on the queue is received.
- If `msgtyp` is greater than 0, the first message of type `msgtyp` is received.
- If `msgtyp` is less than 0, the first message of the lowest type that is less than or equal to the absolute value of `msgtyp` is received.

The argument `msgflg` specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

- If `(msgflg & IPC_NOWAIT)` is "true", the calling-process will return immediately with a return value of -1 and `errno` set to `ENOMSG`.
- If `(msgflg & IPC_NOWAIT)` is "false", the calling-process will suspend execution until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue identifier `msqid` is removed from the system. When this occurs, `errno` is set equal to `EIDRM` and a value of -1 is returned.
  - The calling-process receives a signal that is to be caught. In this case a message is not received and the calling-process resumes execution in the manner prescribed in `SIGNAL(BA_OS)`.

Upon successful completion, the following actions are taken with respect to the data structure associated with `msqid`.

- `msg_qnum` is decremented by 1.
- `msg_lrpid` is set equal to the process-ID of the calling-process.
- `msg_rtime` is set equal to the current time.

**RETURN VALUE**

- If successful, the function `msgsnd` returns a value of 0.
- If successful, the function `msgrcv` returns a value equal to the number of bytes actually placed into the buffer `mtext`.
- Otherwise, the function `msgsnd` and the function `msgrcv` return -1 and `errno` will indicate the error.
The function `msgsnd` will fail and no message will be sent if one or more of the following are true:

- **EINVAL** The value of `msqid` is not a valid message-queue-identifier; or the value of `mtype` is less than 1; or the value of `msgsz` is less than 0 or greater than the system-imposed limit.
- **EACCESS** Operation permission is denied to the calling-process.
- **EAGAIN** The message cannot be sent for one of the reasons cited above and `(msgflg&IPC_NOWAIT)` is "true".
- **EINTR** The function `msgsnd` was interrupted by a signal.
- **EIDRM** The message-queue-identifier `msgid` has been removed from the system.

The function `msgrcv` will fail and no message will be received if one or more of the following are true:

- **EINVAL** The value of `msqid` is not a valid message-queue-identifier; or the value of `msgsz` is less than 0.
- **EACCESS** Operation permission is denied to the calling-process.
- **EINTR** The function `msgrcv` was interrupted by a signal.
- **EIDRM** The message-queue-identifier `msqid` has been removed from the system.
- **E2BIG** The value of `mtext` is greater than `msgsz` and `(msgflg&MSG_NOERROR)` is "false".
- **ENOMSG** The queue does not contain a message of the desired type and `(msgtyp&IPC_NOWAIT)` is "true".

**SEE ALSO**

`MSGCTL(KE_OS)`, `MSGGET(KE_OS)`, `SIGNAL(BA_OS)`.

**LEVEL**

Level 1.
NAME

nice — change priority of a process

SYNOPSIS

int nice(int incr)
int incr;

DESCRIPTION

The function nice adds the value of incr to the nice-value of the calling-process. A process's nice-value is a positive number for which a more positive value results in lower CPU priority.

The system imposes an implementation-specific, maximum process-nice-value of $2 \times \{NZERO\} - 1$ and a minimum process-nice-value of 0. If adding incr to the process's current nice-value would cause the result to be above or below these limits, the process's nice-value will be set to the corresponding limit.

RETURN VALUE

If successful, the function nice returns the process's new nice-value minus \{NZERO\}.

ERRORS

EPERM The function nice will fail and not change the process's nice-value if incr is negative or greater than $2 \times \{NZERO\}$ and the effective-user-ID of the calling-process is not super-user.

SEE ALSO

EXEC(BA_OS).

LEVEL

Level 1.
NAME
plock — lock process, text, or data in memory

SYNOPSIS
#include <sys/lock.h>

int plock(op)
int op;

DESCRIPTION
The function plock allows the calling-process to lock its text segment (text
lock), its data segment (data lock), or both its text and data segments (pro­
cess lock) into memory. Locked segments are immune to all routine swap­
ping. The function plock also allows these segments to be unlocked. The
effective-user-ID of the calling-process must be super-user to use this call.
The argument op specifies the following, which are defined by the
<sys/lock.h> header file:

PROCLOCK lock text and data segments into memory (process lock)
TXTLOCK lock text segment into memory (text lock)
DATLOCK lock data segment into memory (data lock)
UNLOCK remove locks

RETURN VALUE
If successful, the function plock returns 0 to the calling-process; other­
wise, it returns -1 and errno will indicate the error.

ERRORS
The function plock will fail and not perform the requested operation if
one or more of the following are true:

EPERM The effective-user-ID of the calling-process is not super-user.
EINVAL The argument op is equal to PROCLOCK and a process-lock, a
text-lock, or a data-lock already exists on the calling-process.
EINVAL The argument op is equal to TXTLOCK and a text-lock, or a
process-lock already exists on the calling-process.
EINVAL The argument op is equal to DATLOCK and a data-lock, or a
process-lock already exists on the calling-process.
EINVAL The argument op is equal to UNLOCK and no type of lock
exists on the calling-process.

APPLICATION USAGE
The function plock should not be used by most applications. Only pro­
grams that must have the type of real-time control it provides should use it.
SEE ALSO

EXEC(BA_OS), EXIT(BA_OS), FORK(BA_OS).

LEVEL

Level 1.
NAME
profil — execution time profile

SYNOPSIS
void profil(buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;

DESCRIPTION
The argument buff points to an area of memory whose length (in bytes) is
given by bufsiz. After the call to profil, the user's program counter
(pc) is examined each clock tick ([CLK_TCK] times per second); offset is
subtracted from it, and the result multiplied by scale. If the resulting
number corresponds to an entry inside buff, that entry is incremented. An
"entry" is defined as a series of bytes with length sizeof(short).

The scale is interpreted as an unsigned, fixed-point fraction with binary point
at the left: 0177777 (octal) gives a 1-1 mapping of pc's to words in
buff; 0777777 (octal) maps each pair of instruction words together.
02(octal) maps all instructions onto the beginning of buff (producing a
non-interrupting core clock).

Profiling is turned off by giving a scale of 0 or 1. It is rendered
ineffective by giving a bufsiz of 0. Profiling is turned off when an
EXEC(BA_OS) routine is executed, but remains on in both child and parent
after a call to the FORK(BA_OS) routine. Profiling will be turned off if an
update in buff would cause a memory fault.

RETURN VALUE
Not defined.

APPLICATION USAGE
The function profil would normally be used by an application program
only during development of a program to analyze the program's performance.

LEVEL
Level 1.
NAME
ptrace — process trace

SYNOPSIS
int ptrace(request, pid, addr, data)
int request, pid, data;

DESCRIPTION
The function ptrace provides a means by which a parent-process may
control the execution of a child-process. Its primary use is for the implementa-
tion of breakpoint debugging. The child-process behaves normally until it
encounters a signal [see SIGNAL(BA_OS)] at which time it enters a stopped
state and its parent is notified via the WAIT(BA_OS) routine. When the child
is in the stopped state, its parent can examine and modify its core-image
using ptrace. Also, the parent can cause the child either to terminate or
continue, with the possibility of ignoring the signal that caused it to stop.

The data type of the argument addr depends upon the particular
request given to ptrace.

The argument request determines the precise action to be taken by
ptrace and is one of the following:

0  This request must be issued by the child-process if it is to be
traced by its parent. It turns on the child's trace flag that stipu-
lates that the child should be left in a stopped state upon receipt
of a signal rather than the state specified by func [see
SIGNAL(BA_OS)]. The pid, addr, and data arguments are
ignored, and a return value is not defined for this request. Pecu-
liar results will ensue if the parent does not expect to trace the
child.

The remainder of the requests can only be used by the parent-process. For
each, pid is the process-ID of the child. The child must be in a stopped
state before these requests are made.

1, 2  With these requests, the word at location addr in the address
space of the child-process is returned to the parent-process. If
instruction (I) and data (D) space are separated, request 1
returns a word from I-space, and request 2 returns a word from
D-space. If I-space and D-space are not separated either request
1 or request 2 may be used with equal results. The data
argument is ignored. These two requests will fail if addr is not
the start address of a word, in which case a value of -1 is
returned to the parent-process and the parent's errno is set to
EIO.

3  With this request, the word at location addr in the child's
user-area in the system's address space is returned to the parent-
process.
The argument \texttt{data} is ignored. This request will fail if \texttt{addr}
is not the start address of a word or is outside the \textit{user-area}, in
which case a value of \texttt{-1} is returned to the parent-process and
the parent's \texttt{errno} is set to \texttt{EIO}.

With these requests, the value given by the \texttt{data} argument is
written into the address space of the child at location \texttt{addr}. If
I-space and D-space are separated, request 4 writes a word into
I-space, and request 5 writes a word into D-space. If I-space and
D-space are not separated, either request 4 or request 5
may be used with equal results. Upon successful completion, the
value written into the address space of the child is returned to the
parent.

These two requests will fail if \texttt{addr} is a location in a pure pro-
cedure space and another process is executing in that space, or
\texttt{addr} is not the start address of a word. Upon failure a value of
\texttt{-1} is returned to the parent-process and the parent's \texttt{errno}
is set to \texttt{EIO}.

With this request, a few entries in the child's \textit{user-area} can be
written.

The argument \texttt{data} gives the value that is to be written and
\texttt{addr} is the location of the entry. Entries that can be written
are implementation-specific but might include general registers
portions of the \textit{processor-status-word}.

This request causes the child to resume execution. If the \texttt{data}
argument is \texttt{0}, all pending signals including the one that caused
the child to stop are canceled before it resumes execution.

If the argument \texttt{data} is a valid signal number, the child
resumes execution as if it had incurred that signal, and any other
pending signals are canceled. The \texttt{addr} argument must be
equal to \texttt{1} for this request. Upon successful completion, the
value of \texttt{data} is returned to the parent. This request will fail if
\texttt{data} is not \texttt{0} or a valid signal number, in which case a value of
\texttt{-1} is returned to the parent-process and the parent's \texttt{errno}
is set to \texttt{EIO}.

This request causes the child to terminate with the same conse-
quences as the \texttt{EXIT(BA_OS)} routine.

This request is implementation-dependent but if operative, it is
used to request single-stepping through the instructions of the
child.

To forestall possible fraud, the function \texttt{ptrace} inhibits the set-user-ID
facility on subsequent \texttt{EXEC(BA_OS)} routines. If a traced process calls and
\texttt{EXEC(BA_OS)} routine, it will stop before executing the first instruction of the
new image showing signal \texttt{SIGTRAP}.
RETURN VALUE
Upon failure, the function `ptrace` returns -1. Return values on successful completion are specific to the request type (see above).

ERRORS
In general, the function `ptrace` will fail if one or more of the following are true:

- **EIO** The value of `request` is an illegal number. See the summary for each request type above.
- **ESRCH** The argument `pid` identifies a child that does not exist or has not executed a `ptrace` with request 0.

APPLICATION USAGE
The function `ptrace` should not be used by application-programs. It is only used by software debugging programs and it is hardware-dependent.

When the function `ptrace` is used to read a word from the address space of the child-process, `request` 1, 2 or 3, the data read and returned by `ptrace` could have the value -1. In this case, a return value of -1 would not indicate an error.

SEE ALSO
EXEC(BA_OS), SIGNAL(BA_OS), WAIT(BA_OS).

LEVEL
Level 1.
SEMCTL(KE_OS)

NAME

semctl — semaphore-control-operations

SYNOPSIS

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun {
    int val;
    struct semid_ds *buf;
    ushort *array; } arg;

DESCRIPTION

The function semctl provides a variety of semaphore-control-operations as specified by cmd.

The following semaphore-control-operations as specified by cmd are executed with respect to the semaphore specified by semid and semnum. The level of permission required for each operation is shown with each command [see semaphore-operation-permissions in Chapter 9 — Definitions]. The symbolic names for the values of cmd are defined by the <sys/sem.h> header file.

GETVAL Return the value of semval [see Chapter 9 — Definitions].
Requires read permission.

SETVAL Set the value of semval to arg.val.
When this cmd is successfully executed, the semadj value corresponding to the specified semaphore in all processes is cleared.
Requires alter permission.

GETPID Return the value of sempid.
Requires read permission.

GETNCNT Return the value of semncnt.
Requires read permission.

GETZCNT Return the value of semzcnt.
Requires read permission.
The following **cmds** operate on each **semval** in the set of semaphores.

**GETALL**  
Return **semvals** and place into the array pointed to by **arg.array**.  
Requires read permission.

**SETALL**  
Set **semvals** according to the array pointed to by **arg.array**. When this cmd is successfully executed, the **semadj** values corresponding to each specified semaphore in all processes are cleared.  
Requires alter permission.

The following **cmds** are also available:

**IPC_STAT**  
Place the current value of each member of the data structure associated with **semid** into the structure pointed to by **arg.buf**. The contents of this structure are defined in [Chapter 9 — Definitions](#).  
Requires read permission.

**IPC_SET**  
Set the value of the following members of the data structure associated with **semid** to the corresponding value found in the structure pointed to by **arg.buf**:

- **sem_perm.uid**
- **sem_perm.gid**
- **sem_perm.mode /* only low 9-bits */**

This cmd can only be executed by a process that has an effective-user-ID equal to either that of super-user or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with **semid**.

**IPC_RMID**  
Remove the semaphore-identifier specified by **semid** from the system and destroy the set of semaphores and data structure associated with it. This cmd can only be executed by a process that has an effective-user-ID equal to either that of super-user or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with **semid**.

**RETURN VALUE**

If successful, the value **semctl** returns depends on **cmd** as follows:

- **GETVAL**  
  the value of **semval**.

- **GETPID**  
  the value of **sempid**.

- **GETNCNT**  
  the value of **semncnt**.

- **GETZCNT**  
  the value of **semzcnt**.

- **All others**  
  a value of 0.

Otherwise, **semctl** returns **-1** and **errno** indicates the error.
SEMCTL(KE_OS)

ERRORS
The function `semctl` will fail if one or more of the following are true:

EINVAL  The value of `semid` is not a valid semaphore-identifier; or the value of `semnum` is less than 0 or greater than `sem_nsems`; or the value of `cmd` is not a valid command.

EACCES  Operation permission is denied to the calling-process [see Chapter 9 — Definitions].

ERANGE  The argument `cmd` is equal to `SETVAL` or `SETALL` and the value to which `semval` is to be set is greater than the system imposed maximum.

EPERM   The argument `cmd` is equal to `IPC_RMID` or `IPC_SET` and the effective-user-ID of the calling-process is not equal to that of super-user and it is not equal to the value of `sem_perm_gid` or `sem_perm.gid` in the data structure associated with `semid`.

SEE ALSO
`SEMGET(KE_OS)`, `SEMOP(KE_OS)`.  

LEVEL
Level 1.
NAME
semget — get set of semaphores

SYNOPSIS
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key, nsems, semflg)
    key_t key;
    int nsems, semflg;

DESCRIPTION
The function semget returns the semaphore-identifier associated with
key.

A semaphore-identifier with its associated semid_ds data structure and
its associated set of nsems semaphores [see Chapter 9 — Definitions] are
created for key if one of the following are true:

The argument key is equal to IPC_PRIVATE.
The argument key does not already have a semaphore-identifier
associated with it, and (semflg&IPC_CREAT) is "true".

Upon creation, the semid_ds data structure associated with the new
semaphore-identifier is initialized as follows:

In the operation-permissions structure, sem_perm.cuid and
sem_perm.uid are set equal to the effective-user-ID of the
calling-process; while sem_perm.cgid and sem_perm.gid
are set equal to the effective-group-ID of the calling-process.

The low-order 9-bits of sem_perm.mode are set equal to the
low-order 9-bits of semflg.

The variable sem_nsems is set equal to the value of nsems.

The variable sem_otime is set equal to 0 and sem_ctime is
set equal to the current time.

The data structure associated with each semaphore in the set is not
initialized. The function semctl with the command SETVAL or
SETALL can be used to initialize each semaphore.

RETURN VALUE
If successful, the function semget returns a non-negative integer, namely a
semaphore-identifier; otherwise, it returns −1 and errno will indicate the
error.
SEMGET(KE_OS)

ERRORS
The function semget will fail if one or more of the following are true:

EACCESS A semaphore-identifier exists for key, but operation permission as specified by the low-order 9-bits of semflg would not be granted.

EINVAL The value of nsems is either less than or equal to 0 or greater than the system-imposed limit, or a semaphore-identifier exists for the argument key, but the number of semaphores in the set associated with it is less than nsems and nsems is not equal to 0.

ENOENT A semaphore-identifier does not exist for the argument key and (semflg&IPC_CREAT) is "false".

ENOSPC A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system wide would be exceeded.

EEXIST A semaphore-identifier exists for the argument key but ((semflg&IPC_CREAT) && (semflg&IPC_EXCL)) is "true".

SEE ALSO
SEMCTL(KE_OS), SEMOP(KE_OS).

LEVEL
Level 1.
NAME
semop — semaphore operations

SYNOPSIS
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(semid, sops, nsops)
int semid;
struct sembuf *sops;
unsigned nsops;

DESCRIPTION
The function semop is used to automatically perform an user-defined array
of semaphore operations on the set of semaphores associated with the sema-
phore identifier specified by the argument semid.

The argument sops is a pointer to a user-defined array of semaphore-
operation structures.

The argument nsops is the number of such structures in the array.

Each structure, sembuf, includes the following members:

short sem_num; /* semaphore number */
short sem_op; /* semaphore operation */
short sem_flg; /* operation flags */

Each semaphore operation specified by sem_op is performed on the
corresponding semaphore specified by semid and sem_num.

The variable sem_op specifies one of three semaphore operations:

1. If sem_op is a negative integer and the calling-process has alter-
   permission, one of the following will occur:

   • If semval is greater than or equal to the absolute value of
     sem_op, the absolute value of sem_op is subtracted from sem-
     val. Also, if (sem_flg&SEM_UNDO) is "true", the absolute
     value of sem_op is added to the calling-process's semadj value
     for the specified semaphore [see EXIT(BA_OS) in EFFECTS(BA_ENV) in
     Chapter 10 — Environment]. The symbolic name SEM_UNDO is
     defined by the <sys/sem.h> header file.

   • If semval is less than the absolute value of sem_op and
     (sem_flg&IPC_CREAT) is "true", semop will return
     immediately.

   • If semval is less than the absolute value of sem_op and
     (sem_flg&IPC_CREAT) is "false", semop will increment
     the semncnt associated with the specified semaphore and
     suspend execution of the calling-process until one of the following
     conditions occur:
The value of \texttt{semval} becomes greater than or equal to the absolute value of \texttt{sem op}. When this occurs, the value of \texttt{semncnt} associated with the specified semaphore is decremented, the absolute value of \texttt{sem op} is subtracted from \texttt{semval} and, if \texttt{(sem flg\& SEM UNDO)} is "true", the absolute value of \texttt{sem op} is added to the calling-process's \texttt{semadj} value for the specified semaphore.

The \texttt{semid} for which the calling-process is awaiting action is removed from the system [see \texttt{SEMCTL(KE OS)}]. When this occurs, \texttt{errno} is set equal to \texttt{EIDRM}, and a value of \texttt{-1} is returned.

The calling-process receives a signal that is to be caught. When this occurs, the value of \texttt{semncnt} associated with the specified semaphore is decremented, and the calling-process resumes execution in the manner prescribed in the routines defined in \texttt{SIGNAL(BA OS)}.

2. If \texttt{sem op} is a positive integer and the calling-process has alter permission, the value of \texttt{sem op} is added to \texttt{semval} and, if \texttt{(sem flg\& SEM UNDO)} is "true", the value of \texttt{sem op} is subtracted from the calling-process's \texttt{semadj} value for the specified semaphore.

3. If \texttt{sem op} is \texttt{0} and the calling-process has read permission, one of the following will occur:
   
   - If \texttt{semval} is \texttt{0}, the function \texttt{semop} will return immediately.
   
   - If \texttt{semval} is not equal to \texttt{0} and \texttt{(sem flg\& IPC CREAT)} is "true", the function \texttt{semop} will return immediately.
   
   - If \texttt{semval} is not equal to \texttt{0} and \texttt{(sem flg\& IPC CREAT)} is "false", the function \texttt{semop} will increment the \texttt{semzcnt} associated with the specified semaphore and suspend execution of the calling-process until one of the following occurs:

     - The value of \texttt{semval} becomes \texttt{0}, at which time the value of \texttt{semzcnt} associated with the specified semaphore is decremented.

     - The \texttt{semid} for which the calling-process is awaiting action is removed from the system. When this occurs, \texttt{errno} is set equal to \texttt{EIDRM}, and a value of \texttt{-1} is returned.

     - The calling-process receives a signal that is to be caught. When this occurs, the value of \texttt{semzcnt} associated with the specified semaphore is decremented, and the calling-process resumes execution in the manner prescribed in the routines defined in \texttt{SIGNAL(BA OS)}. 

Kernel Extension Definition
RETURN VALUE
If successful, the function `semop` returns 0; otherwise, it returns -1 and `errno` will indicate the error.

ERRORS
The function `semop` will fail if one or more of the following are true for any of the semaphore operations specified by `sops`:

- **EINVAL** The value of `semid` is not a valid semaphore-identifier; or the number of individual semaphores for which the calling-process requests a `SEM_UNDO` would exceed the limit.
- **EFBIG** The value of `sem_num` is less than 0 or greater than or equal to the number of semaphores in the set associated with `semid`.
- **E2BIG** The value of `nsops` is greater than the system-imposed maximum.
- **EACCES** Operation permission is denied to the calling-process [see Chapter 9 - Definitions].
- **EAGAIN** The operation would result in suspension of the calling-process but `(sem_flg&IPC_CREAT)` is "true".
- **ENOSPC** The limit on the number of individual processes requesting a `SEM_UNDO` would be exceeded.
- **ERANGE** An operation would cause a `semval` to overflow the system-imposed limit, or an operation would cause a `semadj` value to overflow the system-imposed limit.
- **EINTR** The function `semop` was interrupted by a signal.
- **EIDRM** The semaphore identifier `semid` has been removed from the system.

Upon successful completion, the value of `sempid` for each semaphore specified in the array pointed to by `sops` is set equal to the process-ID of the calling-process.

SEE ALSO
- EXEC(BA_OS), EXIT(BA_OS), FORK(BA_OS), SEMCTL(KE_OS), SEMGET(KE_OS).

LEVEL
Level 1.
NAME
shmctl — shared-memory-control-operations

SYNOPSIS
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;

DESCRIPTION
The function shmctl provides a variety of shared-memory-control-operations as specified by cmd. The following values for cmd are available:

IPC_STAT Place the current value of each member of the data structure associated with shmid into the structure pointed to by buf. The contents of this structure are defined in Chapter 9 — Definitions.

IPC_SET Set the value of the following members of the data structure associated with shmid to the corresponding value found in the structure pointed to by buf:

shm_perm.uid
shm_perm.gid
shm_perm.mode /* only low 9-bits */

This cmd can only be executed by a process that has an effective-user-ID equal to either that of super-user or to the value of shm_perm.cuid or shm_perm.uid in the data structure associated with shmid.

IPC_RMID Remove the shared memory identifier specified by shmid from the system and destroy the shared memory segment and data structure associated with it. This cmd can only be executed by a process that has an effective-user-ID equal to either that of super-user or to the value of shm_perm.cuid or shm_perm.uid in the data structure associated with shmid.

RETURN VALUE
If successful, the function shmctl returns 0; otherwise, it returns -1 and errno will indicate the error.
The function `shmctl` will fail if one or more of the following are true:

**EINVAL** The value of `shmid` is not a valid shared-memory-identifier; or the value of `cmd` is not a valid command.

**EACCES** The argument `cmd` is equal to `IPC_STAT` and the calling-process does not have read permission [see `shared-memory-operation-permissions` in Chapter 9 — Definitions].

**EPERM** The argument `cmd` is equal to `IPC_RMID` or `IPC_SET` and the effective-user-ID of the calling-process is not equal to that of super user and it is not equal to the value of `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with `shmid`.

**APPLICATION USAGE**

The functions `shmctl`, `shmget`, and `shmat` and `shmdt` are hardware-dependent and may not be present on all systems. The shared memory routines should not be used by applications except when extreme performance considerations require them.

**SEE ALSO**

`SHMGET(KE_O S)`, `SHMOP(KE_O S)`. 

**LEVEL**

Level 1.

Optional: The function `shmctl` may not be present in all implementations of the Kernel Extension.
SHMGET(KE_OS)

NAME
shmget — get shared-memory-segment

SYNOPSIS
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key, size, shmflg)
key_t key;
int size, shmflg;

DESCRIPTION
The function shmget returns the shared memory identifier associated with key.

A shared-memory-identifier and associated data structure and shared-memory-segment of at least size bytes [see Chapter 9 — Definitions] are created for key if one of the following are true:

The argument key is equal to IPC_PRIVATE.

The argument key does not already have a shared-memory-identifier associated with it and (shmflg&IPC_CREAT) is "true".

Upon creation, the data structure associated with the new shared-memory-identifier is initialized as follows:

The value of shm_perm.cuid and shm_perm.uid are set equal to the effective-user-ID of the calling-process.

The value of shm_perm.cgid and shm_perm.gid are set equal to the effective-group-ID of the calling-process.

The low-order 9-bits of shm_perm.mode are set equal to the low-order 9-bits of shmflg.

The argument shm_segsz is set equal to the value of size.

The value of shm_lpid, shm_nattch, shm_atime, and shm_dtime are set equal to 0.

The value of shm_ctime is set equal to the current time.

RETURN VALUE
If successful, the function shmget returns a non-negative integer, namely a shared-memory-identifier; otherwise, it returns -1 and errno will indicate the error.
ERRORS

The function `shmget` will fail if one or more of the following are true:

**EINVAL** The value of `size` is less than the system-imposed minimum or greater than the system-imposed maximum, or a shared-memory-identifier exists for the argument `key` but the size of the segment associated with it is less than `size` and `size` is not equal to 0.

**EACCES** A shared-memory-identifier exists for `key` but operation permission as specified by the low-order 9-bits of `shmflg` would not be granted.

**ENOENT** A shared-memory-identifier does not exist for the argument `key` and `(shmflg&IPC_CREAT)` is "false".

**ENOSPC** A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system wide would be exceeded.

**ENOMEM** A shared memory identifier and associated shared memory segment are to be created but the amount of available physical memory is not sufficient to fill the request.

**EEXIST** A shared-memory-identifier exists for the argument `key` but `((shmflg&IPC_CREAT) && (shmflg&IPC_EXCL))` is "true".

APPLICATION USAGE

The functions `shmctl`, `shmget` and `shmat` and `shmct1` are hardware-dependent and may not be present on all systems. The shared memory routines should not be used by applications except when extreme performance considerations require them.

SEE ALSO

`SHMCTL(KE_OS)`, `SHMOP(KE_OS)`.

LEVEL

Level 1.

Optional: The function `shmget` may not be present in all implementations of the Kernel Extension.
SHMOP(KE_OS)

NAME
shmop — shared-memory-operations

SYNOPSIS
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat(shmid, shmaddr, shmflg)
int shmid;
char *shmaddr
int shmflg;
int shmdt(shmaddr)
char *shmaddr

DESCRIPTION
The function shmat attaches the shared-memory-segment associated with the shared-memory-identifier specified by shmid to the data segment of the calling-process. The segment is attached at the address specified by one of the following criteria:

If shmaddr is equal to 0, the segment is attached at the first available address as selected by the system.

If shmaddr is not equal to 0 and (shmflg&.SHM_RND) is "true", the segment is attached at the address given by (shmaddr-(shmaddr%SHMLBA)). The character % is the C language modulus operator.

If shmaddr is not equal to 0 and (shmflg&SHM_RND) is "false", the segment is attached at the address given by shmaddr.

The segment is attached for reading if (shmflg&SHM_RDONLY) is "true" and the calling-process has read permission; otherwise, if it is not true and the calling-process has read and write permission, the segment is attached for reading and writing.

The function shmdt detaches from the calling-process's data segment the shared memory segment located at the address specified by shmaddr.

The following symbolic names are defined by the <sys/shm.h> header file:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHMLBA</td>
<td>segment low boundary address multiple</td>
</tr>
<tr>
<td>SHM_RDONLY</td>
<td>attach read-only (else read-write)</td>
</tr>
<tr>
<td>SHM_RND</td>
<td>round attach address to SHMLBA</td>
</tr>
</tbody>
</table>

Page 292

Kernel Extension Definition
RETURN VALUE

If successful, the function `shmat` will return the data segment start address of the attached shared-memory-segment. If successful, the function `shmdt` will return a value of 0. Otherwise, the function `shmat` and the function `shmdt` will return -1 and `errno` will indicate the error.

ERRORS

The function `shmat` will fail and not attach the shared-memory-segment if one or more of the following are true:

- **EACCES** Operation permission is denied to the calling-process [see Chapter 9 - Definitions].
- **ENOMEM** The available data space is not large enough to accommodate the shared memory segment.
- **EINVAL** The value of `shmid` is not a valid shared-memory-identifier; or the value of `shmaddr` is not equal to 0 and the value of `(shmaddr - (shmaddr % SHMLBA))` is an illegal-address; or the value of `shmaddr` is not equal to 0, `(shmflg & SHM_RND)` is "false" and the value of `shmaddr` is an illegal-address.
- **EMFILE** The number of shared-memory-segments attached to the calling-process would exceed the system-imposed limit.

The function `shmdt` will fail and not detach the shared-memory-segment if the following is true:

- **EINVAL** `shmaddr` is not the data segment start address of a shared-memory-segment.

APPLICATION USAGE

The functions `shmctl`, `shmget`, `shmat`, and `shmdt` are hardware dependent and may not be present on all systems. The shared memory routines should not be used by applications except when extreme performance considerations require them.

SEE ALSO

`EXEC(BA_OS)`, `EXIT(BA_OS)`, `FORK(BA_OS)`, `SHMCTL(KE_OS)`, `SHMGET(KE_OS)`.

LEVEL

Level 1.

Optional: the functions `shmat` and `shmdt` may not be present in all implementations of the Kernel Extension.
Appendix

Changes from Issue 1
Appendix

Changes from Issue 1

Only substantive changes from Issue 1 to Issue 2 of the *System V Interface Definition* are described here. Changes that did not alter meaning, for example when text was changed or added for clarity, are not listed below.

Changes in the organization or general changes in the content of the SVID are described first. Summaries of changes in the detailed component definitions follow.
12.1 BASE SYSTEM DIFFERENCES

Organization. The information in the definition of the Base System is ordered somewhat differently in Issue 2. For example, error conditions, environmental variables, data files, directory tree structure, and special device files appear together in Chapter 5 — Environment, in Issue 2. Signals appear in the definition of the function SIGNAL(BA_OS) in Chapter 6 — Operating System Service Routines.

Omissions. Section 2.6 on Header Files in Issue 1 has been omitted. Issue 1 specified that the header files were not expected to be present on an implementation of the Base System. However, the presence of the header file Section was misinterpreted by many to mean that these files were part of the Base System. In Issue 2, all necessary information about a header file appears in the detailed definitions of those routines that use the header file.

Appendix 1.6 in Issue 1, Comparison to the 1984 /usr/group Standard, was not carried over to Issue 2 because the work of the /usr/group committee has been subsumed by the IEEE P1003 working group.

The routines regcmp and regex were mistakenly included in Issue 1. Issue 2 removed and replaced these routines with the routines defined in REGEXP(BA_LIB).

The names of three external variables, errno, sys_errlist and sys_nerr, mistakenly appeared in the list of library routine names in Section 2.5 of Issue 1. They have been removed from the corresponding table in Issue 2.

Future Directions. Issue 2 made some additions to Chapter 2 — Future Directions; these are not detailed in this summary.

The paragraphs below identify specific changes to detailed component definitions:

Definitions.

special-processes

Issue 1 specified that process-IDs 0 and 1 were reserved for special-processes. To allow implementations to reserve more ID numbers for special-processes, Issue 2 specifies that at least these two IDs are reserved.

Environment.

ERRORS(BA_ENV).

Issue 2 additionally specifies that no error condition will have the value zero.

In Issue 2 the EFAULT error condition, when an address is outside the address space of a process, is not required from all systems [see ERRNO(BA_ENV)].

Issue 2 additionally specifies that errno should not be checked unless an error is indicated by a routine.

FILSYS(BA_ENV).

Issue 1 incorrectly specified that all the environmental variables required to be set in the Base System environment were defined by the /etc/profile file. Issue 2 specifies that the /etc/profile file may define the
variables *PATH* and *TZ*.

Issue 2 removed the description of an encrypted password from the definition of the `/etc/passwd` file.

**TERMIO(BA_ENV).**

Issue 1 incorrectly listed the commands **TCGETA** and **TCSETA** as **TCGETS** and **TCSETS** in the definition of **TERMIO(DEV)**. Issue 2 lists them as **TCGETA** and **TCSETA** in **TERMIO(BA_ENV)**.

Issue 2 eliminated the **APPLICATION USAGE** section.

**OS Service Routines.**

**CHMOD(BA_OS).**

Issue 1 identified the access permission bit 01000 as "save text image after execution"; Issue 2 identifies it as "reserved".

**CHOWN(BA_OS).**

Issue 1 read "if chown is invoked by other than the super-user, the set-user-ID and the set-group-ID bits of the file mode will be cleared." Issue 2 clarifies that chown must be *successfully* invoked by other than super-user for this to occur.

**CREAT(BA_OS).**

Issue 1 mistakenly stated that corresponding access-permission bits of the file mode were ANDed with the process’s file-mode creation mask. Issue 2 correctly specifies that corresponding access-permission bits of the file mode are ANDed with the complement of the process’s file-mode creation mask.

**EXIT(BA_OS).**

Issue 2 specifies that termination of a process by exiting does not terminate its children.

Issue 1 mistakenly stated that the **SIGHUP** signal is sent to each member of a process-group if the calling-process is a process-group-leader. Issue 2 specifies that the calling-process must also be associated with a control terminal.

**FCNTL(BA_OS).**

Issue 2 notes that the function **fcntl** commands **F_GETLK**, **F_SETLK** and **F_SETLKW** are post-System V Release 2.0 features.

Issue 2 clarifies when a read-lock or a write-lock can be set on a file with existing locks.

Issue 1 incorrectly specified the **_sys_id** element in the **flock** structure. It was removed in Issue 2.

Issue 1 incorrectly specified that the **EDEADLK** error condition would occur when the **fcntl** command was **F_SETLK** and putting the process to sleep would cause a deadlock. Issue 2 correctly specifies that the command in this case is **F_SETLKW**.
In Issue 1, the error condition EAGAIN should have been EACCES. This was changed in Issue 2 and the migration to EAGAIN is shown in FUTURE DIRECTIONS.

Issue 2 recommends that applications test for errno equal to either EAGAIN or EACCES.

**FOPEN(BA_OS).**

Issue 2 additionally specifies that the functions fopen and freopen will fail if the argument type is invalid or the file cannot be opened; the function fdopen will fail if the argument type is invalid or the argument fildes is not an open file-descriptor; fopen and freopen will fail if there are no free stdin streams available.

Issue 1 specified that if the argument path could not be accessed by the functions fopen and freopen, errno could contain any of the values listed for the function open. Issue 2 further specifies which of these errno values are possible.

**GETCWD(BA_OS).**

Issue 1 specified a side-effect if the function were called with a null pointer. Issue 2 removed this side-effect from the definition.

**IOCTL(BA_OS).**

Issue 2 additionally specifies that the data type of the argument arg is either an integer or a pointer to a device-specific data structure.

**LOCKF(BA_OS).**

Issue 1 misspelled the F_UNLOCK value of the argument function as F_UNLOCK in one place.

Issue 1 incorrectly specified that the error condition EDEADLK would occur if the argument cmd was F_LOCK or F_TLOCK and a deadlock would occur. Issue 2 correctly specifies that the argument cmd was F_LOCK and a deadlock would occur.

In Issue 1, the error condition EAGAIN should have been EACCES. This was changed in Issue 2 and the migration to EAGAIN is shown in FUTURE DIRECTIONS.

Issue 2 recommends that applications test for errno equal to either EAGAIN or EACCES.

**LSEEK(BA_OS).**

Issue 2 removed the reference to the SIGSYS signal on the error condition EINVAL.

**MALLOC(BA_OS).**

Issue 1 incorrectly specified that the argument value to the function malloc must be greater than 0 when the argument command is equal to M_NBLKKS. Issue 2 correctly specifies that value must be greater than 1.

Changes from Issue 1
Changes from Issue 1

Issue 2 additionally specifies that the function `malloc` must not be called until after some storage has been allocated using the function `malloc`. Issue 2 additionally specifies that the functions `malloc`, `calloc` and `realloc` will fail if `nbyte` is 0.

**MKNOD(BA_OS).**

Issue 1 identified the access permission bit `01000` as "save text image after execution"; Issue 2 identifies it as "reserved". For the error condition `EACCES`, Issue 2 additionally specifies that the effective-user-ID of the process is not super-user.

**MOUNT(BA_OS).**

Issue 2 adds a new **FUTURE DIRECTIONS** section.

**OPEN(BA_OS).**

Issue 1 mistakenly stated that corresponding access-permission bits of the file mode were ANDed with the process's file-mode creation mask. Issue 2 correctly specifies that corresponding access-permission bits of the file mode are ANDed with the complement of the process's file-mode creation mask.

Issue 2 additionally specifies that the new file-descriptor returned is the lowest numbered file-descriptor available.

**READ(BA_OS).**

Issue 2 added the error conditions `EIO` and `ENXIO` which were mistakenly omitted in Issue 1.

Issue 2 additionally specifies that reading from a section of a file to which no data were written will read bytes with value zero into the buffer.

**SETUID(BA_OS).**

Issue 2 removed references to the saved set-group-ID because this is not a feature in System V Release 1.0 or Release 2.0.

**SYSTEM(BA_OS).**

Issue 2 removed references to positional parameters. Issue 2 also removed the paragraph on "here" documents, `<<[−]word`, which was incorrectly included in Issue 1.

**TIME(BA_OS).**

Issue 1 read "As long as the argument `tloc` is not zero, the return value is also stored in the location to which the argument `tloc` points." Issue 2 reads "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."

**WRITE(BA_OS).**

Issue 2 specifies in more detail the run-time behavior when writing to a pipe, particularly atomic and partial writes.

Issue 2 added the error conditions `EIO` and `ENXIO`, which were mistakenly omitted in Issue 1.
UTIME(BA_OS).

Issue 2 clarifies that the utimebuf structure must be defined by the user.

WAIT(BA_OS).

Issue 2 removed all references in the Base System to a child-process stopping as a result of being traced because this functionality applies to the Kernel Extension.

General Library Routines.

CRYPT(BA_LIB).

Issue 2 made the functions crypt, setkey and encrypt optional because U.S. State Department regulations may restrict the export of these routines. If present on an implementation, each routine's source-code interface and run-time behavior is expected to conform to the definition.

CTIME(BA_LIB).

Issue 2 removed the include statement for the <sys/types.h> header file, which was mistakenly included in the SYNOPSIS section.

EXP(BA_LIB).

The RETURN VALUE section in Issue 1 read "log and log10 will return HUGE". Issue 2 corrected this to read "log and log10 will return -HUGE".

The FUTURE DIRECTIONS section in Issue 1 read "log and log10 will return -HUGE_VAL when n is not positive" and "sqrt will return -0 when the value of n is -0". Issue 2 corrected this to read "log and log10 will return -HUGE_VAL when x is not positive" and "sqrt will return -0 when the value of x is -0".

FTW(BA_LIB).

Issue 1 specified that if the value of the argument depth were zero or negative, the effect was the same as if the value were 1. Issue 2 specifies that the value of depth should be in the range of 1 to (OPEN_MAX).

GAMMA(BA_LIB).

The RETURN VALUE paragraph in Issue 1 read "For non-negative integer arguments, HUGE is returned". Issue 2 corrected this to read "For non-positive integer arguments, HUGE is returned".

HSEARCH(BA_LIB).

Issue 2 removed the second paragraph of the APPLICATION USAGE section because it applies to developing an application-program using HSEARCH(BA_LIB), and it does not apply to an executable program that uses HSEARCH(BA_LIB).

PERROR(BA_LIB).

Issue 2 additionally specifies the behavior of the function when the argument is a null string.

PRINTF(BA_LIB).

Issue 2 additionally specifies escape sequences that may be used in the format argument.

Issue 2 removed the conversion character i because it is not available in System V Release 1.0 or Release 2.0.
PUTC(BA_LIB).  Issue 2 specifies the return value of the function putw; Issue 1 did not.

PUTENV(BA_LIB).  Issue 1 failed to specify that this routine first became available with System V Release 2.0.

SCANF(BA_LIB).  Issue 2 removed the conversion characters i and n because they are not available in System V Release 1.0 or System V Release 2.0.

STRING(BA_LIB).  Issue 2 specifies that the function strtok will write a null character into the string s1 immediately following a matched token.

Issue 2 removed the word "optional" from the first sentence of the APPLICATION USAGE section.

UNGETC(BA_LIB)  The special case for stdin that appeared in Issue 1 was removed from the definition in Issue 2.

12.2 KERNEL EXTENSION DIFFERENCES

Definitions.

sem structure  Issue 1 incorrectly included two elements, semnwait and semzwait, in the structure sem; Issue 2 removed them.

EFFECTS(KE_ENV)  Issue 2 added effects on EXEC(BA_OS) and FORK(BA_OS) routines that were mistakenly omitted in Issue 1.

Issue 2 specifies additional values of errno for the Kernel Extension that were omitted in Issue 1.

ACCT(KE_OS).  Issue 1 incorrectly specified the type of the element acct of the structure acct as time_t; Issue 2 specifies its type as comp_t.

Issue 2 specifies the values of the fields ac_flag and ac_comm that result from a call to an EXEC(BA_OS) routine or the FORK(BA_OS) routine.

OS Service Routines.

MSGCTL(KE_OS).  Issue 2 specifies that read permission is needed for the IPC_STAT command.

NICE(KE_OS).  Issue 2 replaced the constant 39 with the expression 2*(NZERO)-1 to indicate that this is an implementation-specific constant.

PTRACE(KE_OS).  Issue 1 specified the type of the argument addr as int. Issue 2 specifies that the type of the argument addr is dependent upon the value of the argument request.

SEMCTL(KE_OS).  Issue 2 specifies the permission level needed for semctl operations.
Issue 1 incorrectly specified the type of the argument `sops` as `struct sembuf **`; Issue 2 specifies the type as `struct sembuf *`.

Issue 1 incorrectly specifies the type of the argument `nsops` as `int`; Issue 2 specifies the type as `unsigned`.

Issue 2 specifies the permission level needed for `sem_op` operations.

Issue 1 incorrectly specified the successful return value of the function. Issue 2 specifies that the function will return 0 on success. Issue 2 removed the first paragraph of the `RETURN VALUE` section.

Issue 1 incorrectly included the commands `SHM_LOCK` and `SHM_UNLOCK`. Issue 2 removed these and the two error conditions that referenced them.

Issue 2 specifies that read permission is needed for the `IPC_STAT` command.

Issue 1 omitted from the description of the `EPERM` error condition that the effective-user-ID was not equal to `shm_perm.cuid`; Issue 2 added this.
Indexes
standard error output 32, 36, 45, 82, 153, 170, 176, 196, 219, 236-237
error recovery 197
error, last error encountered 87, 196
error-handling procedures 153, 171, 176, 187, 190, 223, 237
escape character 46-47, 54-55
ESPIPE 40, 95, 101
ESRCH 40, 95, 99, 279
ETXTBSY 40, 59, 68, 72, 83, 112, 142
Euclidean distance function 187
evolution of the SVID, mechanism for 6-7
exception handling mechanisms, IEEE 9
EXDEV 40, 97
executable file 19, 43, 70, 135
execute mode 59
execute/search permission 27
execution time 103, 221, 276
execution time profile 276
exit status 115, 131-132, 135
exiting a process 6, 20-21, 32, 58, 72-74, 115, 123, 131-132, 135, 146-147, 206, 212, 258, 262-263, 275, 278, 285, 287, 293, 299
external variable 37, 109, 162, 174, 180, 196, 213, 260, 298
file
access 19, 21, 27-28, 36-37, 59-60, 63-64, 67-68, 83, 111, 144-145, 264
access time 144-145
append mode 75, 82-83, 110, 134, 148
block special 32, 39, 106, 138
change group 65
change mode 63
change owner 19, 63, 65
character special 28, 39
close 28, 58, 66, 68, 101, 112, 117, 150
close-on-exec flag 71, 75, 85
create new link 97
end of 77, 82-83, 89, 99, 102, 110, 116, 127, 148, 217, 244
execute 19, 35, 39, 42, 59, 68, 70, 77, 84, 112, 115, 126, 142, 174
execute permission 39, 59, 112, 126
fifo special 112
file descriptor 28, 38, 134, 149, 242
file pointer 28, 83, 116-117, 148-149, 177, 204
file pointer inherited 71, 110-111
file pointer initialization 67
file pointer reposition 80, 89, 101
flags 28, 53, 75, 77, 110-111, 114, 148, 174
group 27, 36, 59, 63, 65, 67, 106, 111, 139
last access time 127
link 27, 29, 34, 38-40, 97, 127, 142
locks 66, 68, 76-79, 85, 98-100, 299
maximum size 38, 149
mode 27-28, 45, 59, 63-65, 67-68, 72, 83, 106, 111, 115, 127, 139, 299, 301
mode creation mask 106, 111, 139, 299, 301
mode permission bits 59, 64, 67-68, 106, 111, 139
modification time 144
open a 82-84, 110-112, 114-115, 300-301
ordinary 28, 39-40, 43, 63, 65, 67, 70, 72, 263
owner 19, 27, 40, 59, 63-65, 67, 106, 111, 139, 144-145
permissions 27, 39, 44, 59, 63-64, 67-68, 72, 83, 106, 108, 111-112, 126, 139, 144, 280
pipe 75, 114, 116, 127
pure procedure shared text file 59, 68, 72, 84, 112, 142
read-only 40, 59, 64-65, 68, 83, 107, 111, 142, 145, 263
reading 35, 53, 112, 114-117, 177, 183, 301
remove 66, 76-77, 98, 142, 235, 299, 302
rewrite 67
set status flags 75, 110

System V Interface Definition
size limit 138, 148-149
status 28, 75, 110, 126-127, 144, 148
status flags 28, 75, 110, 148
stream 32, 74, 80, 82-83, 89, 115, 177, 204, 217, 219, 233, 243, 300
truncate 67, 111, 134
unlink 127, 142, 235
update 66, 78, 82-83, 89, 144, 233
writing 32, 36-37, 39, 67-68, 72, 74, 82, 108, 111, 114-115, 148-149, 204, 207
file access permissions 27, 59, 63-64, 67-68, 111, 144
file descriptor 28, 39, 134, 149, 242
file descriptors, maximum open 39
file system
entry mount point 142
mount 108, 140, 142-143
read-only 40, 59, 64-65, 68, 84, 107, 111, 142, 263
unmount 140
file table 39, 68, 112, 184
file-descriptor, duplicate 69
file-descriptor, get a new open 69, 110-112, 114
file-descriptor-1 134
file-descriptor-2 134
file-name 28-30, 44, 72, 131, 160, 234-235
file-name expansion 28
fill-characters 50-51
fixed-point fraction 276
floating point 9, 122, 166, 172-173, 201, 203, 216, 218, 229
standards 9, 203, 218
fractional time-zones 41, 163
fraud 278
full-duplex mode 45
function address, catch signal 123
F_SETLK 76-78, 299
F_SETLKW 76-78, 299
F_TLOCK 98-99, 300
F_UNLOCK 98-99, 300

G

general terminal interface 45, 93-94

System V Interface Definition
null character 28, 30, 133, 135, 143, 183, 193, 199, 202, 207, 213, 226-227, 230, 234, 303
null special file 35

O

OCRNL 50
OFDEL 50
OLCUC 50
ONLRET 50
ONOCR 50
open stream 9, 32, 74, 80, 82-83, 89, 115, 204, 219, 233, 300
Open Systems Networking Interfaces 9
operating system services 5-7, 9, 19-21, 57-150, 249, 258, 298
operation permission 252-253, 255-256, 268, 272, 280, 282-284, 287, 291, 293, 304
OPOST 50
ordinary file 28, 39-40, 43, 63, 65, 67, 70, 72, 263
OSI, Open Systems Interconnection reference model 9
output control value 52
owner 19, 27, 40, 59, 63-65, 67, 71, 106, 111, 139, 144-145

System V Interface Definition
set file status flags 75, 110, 148
set system time 11, 38, 129
set-user id 27, 63, 65, 120, 278, 299
shared-memory
applications 289, 291, 293
identifier 255-256, 258, 288-293
segment 255-256, 258, 288, 290-293
segments detached 292
segment 255, 288, 291-293
shared-memory 252, 288-293
shared-memory identifier 288-293
shared-memory segment, address 292-293
shared-memory segment, size 291
shared resource environment 9
shell 4, 135
SIGALRM 61, 122
SIGFPE 122
SIGHUP 47, 122, 134
SIGILL 122-123
SIGINT 122, 134
SIGKILL 95-96, 122-124
signal
abort 58, 124
alarm 61, 72, 86, 99, 113, 125
default action 71, 122, 224
ignore 40, 46-47, 58, 71, 113, 123,
135, 224, 277
interrupt 38, 45-46, 49, 99, 123, 272,
287
kill 30, 38, 95, 113, 119, 124
quit 38, 45-46
receipt 122-123, 146, 277
sending 58, 61, 95, 124
signal handling 45, 122
signal number, illegal 124
signal-catching function 113, 123, 125
signals 20, 30, 32, 38, 40, 45-47, 49, 52-
53, 56, 58, 61, 71-73, 83, 86, 91, 93,
95-96, 99, 112-113, 117, 119, 122-
125, 134, 146-147, 149, 222, 224-225,
262-263, 272, 277-279, 286-287,
298-300
SIGPIPE 122, 149
SIGQUIT 122, 134
SIGSYS 122, 300
SIGTERM 122, 124
SIGTRAP 122-123, 278
SIGUSR1 122, 124
SIGUSR2 122, 124
SIG_DFL 71, 123, 224
SIG_IGN 71, 123, 224
simple-command 131, 133-135
SING, math argument singularity error
170, 176, 190
small-block memory allocation 104
Software Development Extension 4, 24
software signal 122, 224
source-code interfaces 4-7, 19, 21, 24, 249,
302
special device files 5, 19, 40, 42, 112, 298
special file 5, 19, 28, 34-35, 40, 42, 45, 48,
112, 116, 177, 204, 298
special file, /dev/null 35
special system processes 31, 146
SS2 character 13
SS3 character 13
standard error 11, 32, 36, 45, 82, 153,
170, 176, 196-198, 219, 236-237
standard error, stream stderr 32, 82, 219
Standard I/O routines 31
standard input 32, 36, 45, 115, 131, 133-
134, 182-183, 215
Standard Input/Output 21
standard output 32, 36, 45, 82, 115, 131-
134, 153, 170, 176, 196, 199, 207,
219, 236-237
START/STOP 46, 49, 50
start/stop output control 49
stderr 32, 36, 82, 180, 219
stdin 32, 36, 82, 177, 183, 215, 303
stdio 21, 31-32, 36, 66, 68, 80, 83, 100-
101, 112, 117, 150, 160, 164, 219,
234, 300
stdio stream 32, 74, 80, 82-83, 87, 89,
115, 167, 177, 183, 199, 204-205,
207, 215, 217, 219-220, 233, 243, 300
close 74, 220
open 9, 32, 74, 80, 82-83, 89, 115,
204, 219, 233, 300
reposition file pointer 80, 89
stdio, header file 31-32, 160, 164, 219,
234
stdio, routines 21, 31-32, 66, 68, 80, 83,
100-101, 112, 117, 150
stdio, stream definition 32
stdout 32, 36, 82, 199, 207
stopped state, child 277
streams I/O interfaces for networking 9
string manipulation routines 23
string operations 23, 226
super-user 27, 32, 40, 63-65, 95, 97, 106-
108, 120, 129, 138, 140, 142, 144-
145, 253, 255-256, 262-266, 273-274,
281-282, 288-289, 299, 301
suspend a process 47, 124, 254
suspend execution 125, 285-286

System V Interface Definition
Function Index

A
abort function 20-21, 58, 124
abs function 22, 152, 172
access function 20, 59-60
acct function 249, 258, 262-263, 303
acos function 22, 236-237
advance function 22, 210, 213-214
alarm function 20, 61, 72, 86, 99, 113, 122, 125
asctime function 22, 161-162
asin function 22, 236-237
atan function 22, 236
atan2 function 22, 236-237
atof function 22, 229
atoi function 22, 230
atol function 22, 230
B
bsearch function 22, 155, 186, 189, 208, 241
C
calloc function 5, 20, 103-105, 301
ceil function 22, 172
cbdif function 20, 39, 62, 264
chmod function 20, 27, 60, 63-65, 67-68, 71, 100, 107, 111, 127-128, 139, 299
chown function 19-20, 63-65, 127-128, 299
chroot function 249, 264
clearerr function 20, 31, 80
clock function 22, 157
close function 20-21, 28, 66, 68-69, 74, 79, 100-101, 112, 117, 142, 150
compile function 210, 213-214
conv routines 22-23, 158
cos function 22, 236
cosh function 22, 223
creat function 20, 28, 66, 67-69, 84, 100, 102, 111-112, 117, 127-128, 139, 150, 233, 235, 299
crypt function 22, 44, 159, 302
ctermid function 22, 31, 160
ctime function 22-23, 127, 144, 161, 163, 253-254, 283, 290, 302
ctype routines 11, 22, 158, 164, 229-231

D
drand48 function 22-23, 166-167, 209
dup function 20, 28, 66, 68-69, 84, 102, 112, 117, 135, 150

e
encrypt function 22, 159, 299, 302
erand48 function 22, 166-167
erf function 22, 169
erfc function 22, 169
execl function 20, 70-72
execlp function 20, 70-72
execle function 20, 70-72
execvp function 20, 70-72
execve function 20, 70-72
execvpe function 20, 70-72

E
ext function 20-21, 32, 58, 72, 73, 74, 123, 146-147, 258, 262-263, 275, 278, 285, 287, 293, 299
exp function 22-23, 169, 170, 171, 173, 176, 302

F
fabs function 22, 172
fclose function 20, 31, 66, 73-74, 84, 115, 178, 205
fcntl function 20, 28, 66-69, 71, 75-78, 85-86, 98-100, 102, 110-112, 117, 299-300
dopen function 20, 31, 82-83, 300
feof function 20, 31, 80
ferror function 20, 31, 80, 87
ferror function 20, 31, 80, 87-88, 177-178, 183, 205, 207
fopen function 20, 31, 82-83, 300
fget function 20, 31, 80, 87
fget function 20, 31, 80, 87-88, 177-178, 183, 205, 207
fflush function 20, 31, 74
fgets function 22, 31, 177
fgets function 22, 31, 183
fileno function 20, 31, 80
floor function 22, 152, 172
fmopen function 22, 172
fork function 20-21, 29-30, 37, 39, 45, 55, 61, 72, 77, 85-86, 91, 119, 135, 137, 147, 259, 262, 275-276, 287, 293, 303

System V Interface Definition
fprintf function 22, 31, 199, 202, 244
fputc function 22, 31, 204-205
fputs function 22, 31, 207
fread function 20, 31, 87-88, 100, 117, 178, 183, 205, 207
free function 20, 103, 234
freopen function 20, 31, 82-83, 300
frexp function 22, 173
fscanf function 22, 31, 215
fseek function 20, 31, 80, 83-84, 87-89, 101, 115, 243
fstat function 20, 126-128
ftell function 20, 31, 89
fwrite function 20, 31, 87-88, 100, 150

gamma function 22, 176, 302
getc function 22, 31, 88, 158, 177, 183, 205, 218, 220, 243
getchar function 22, 31, 177, 218
getcwd function 20, 90, 300
getgid function 20, 92
getenv function 22, 163, 179, 206
getuid function 20, 92
getgid function 20, 92
getopt function 22, 180-182
getgpgp function 20, 91
getpid function 20, 91, 96, 119, 195, 280-281
getppid function 20, 91
gets function 22, 31, 88, 177-178, 183
getuid function 20, 92, 121
getw function 22, 31, 177, 205
gmtime function 22, 161, 163
gsignal function 22, 224

hcreate function 22, 184-185
hdestroy function 22, 184
hsearch function 22, 156, 184-185, 189, 241, 302
hypot function 22, 171, 187

ioctl function 20, 28, 39, 47-48, 55, 93, 116-117, 123, 300
isalnum function 22, 164-165
isalpha function 22, 164
isascii function 22, 164-165

isatty function 22, 242
iscntrl function 22, 164-165
isdigit function 22, 164
isgraph function 22, 164-165
islower function 22, 164
isprint function 22, 164-165
isspace function 22, 164-165
isupper function 22, 164
isxdigit function 22, 164-165

j0 function 22, 153
j1 function 22, 153
jn function 22, 153
jrand48 function 22, 166-167

kill function 20, 30, 38, 40, 45-47, 54, 95, 113, 119, 122, 124

lcong48 function 22, 166-167
ldexp function 22, 2173
lfind function 22, 188
link function 20, 27, 29, 34, 38-40, 97, 127-128, 142
localtime function 22, 161, 163
lockf function 20-21, 79, 85-86, 98-99, 127, 300
log function 22, 43, 170-171, 176, 302
log10 function 22, 170-171, 302
longjmp function 22, 221
lrand48 function 22, 166-167
lsearch function 22, 156, 186, 188-189, 208, 241
lseek function 20-21, 40, 68, 80, 89, 101-102, 112, 116, 150, 300

malloc function 20, 103, 105, 301
mallopt function 20, 103-105, 175, 185-186, 206, 220, 234-235, 300, 301
makedev function 20, 103-105, 300
matherr function 22, 193
memcmp function 22, 193
memcmp function 22, 193-194
memcpy function 22, 193
memset function 22, 193
mknod function 20, 64, 72, 106-107, 127-128, 139, 301
mktemp function 22, 195, 233, 235
modf function 22, 173
mount function 20, 37-40, 108-109, 140, 142-143, 145, 301
mrand48 function 22, 166-167
msgctl function 249, 253, 260, 265-266, 268, 272, 303
msgget function 249, 252, 266, 267, 268, 272
msgrcv function 249, 253, 269-272
msgsnd function 249, 253, 269, 271-272

nice function 249, 273, 303
nrand48 function 22, 166-167


pause function 20, 61, 113, 123-125, 147
pclose function 20, 31, 115
perror function 22, 196, 302
pipe function 20, 28-29, 40, 66, 69, 75, 84, 101, 114-117, 122, 127-128, 131, 135, 148-150, 302
plock function 249, 258, 274
popen function 20, 31, 89, 115, 117
pow function 22, 170-171

qsort function 22, 155-156, 208

rand function 22-23, 168, 209
read function 47, 54, 80, 87, 101-102, 114, 115, 116, 117, 123, 183, 205, 215, 219, 270, 279
realloc function 20, 103-105, 301
rewind function 20, 31, 80, 83, 89

seed48 function 22, 166-168
semctl function 249, 254-255, 260, 280-284, 286-287, 303
semget function 249, 254, 282, 283, 284, 287
semop function 249, 254-255, 258, 282, 284, 285, 286, 287, 304
setbuf function 22-23, 31-32, 74, 205, 219, 243
setgid function 20, 27, 30, 120
setjmp function 22, 124, 221
setkey function 22, 159, 302
setpgid function 20, 30, 45, 55, 91, 96, 119
setuid function 20, 27, 30, 71, 92, 120, 301
setvbuf function 23, 31, 219-220
shmat function 249, 256, 289, 291, 292, 293
shmctl function 249, 256, 260, 281, 288-289, 291, 293, 304
shmget function 249, 256, 289, 291, 292, 293

sin function 23, 236
sinf function 22-23, 171, 223
sleep function 20, 55, 76, 78, 98-99, 125, 300
sprintf function 23, 199, 202, 244

System V Interface Definition
sqrt function 23, 170-171, 187, 302
srand function 23, 209
srand48 function 23, 166-167
sscanf function 23, 215
ssignal function
ssignal function 22-23, 224
stat function 20, 126-128, 144, 146, 174
step function 22, 210, 213-214
stime function 20, 129, 136-137, 253
strcat function 23, 226-227
strchr function 23, 226-227
strcmp function 23, 184, 188, 226-228
strcpy function 23, 226-227
strcspn function 23, 226-227
strlen function 23, 226-228
strncat function 23, 226-228
strncmp function 23, 226-228
strncpy function 23, 226-228
strpbrk function 23, 226-227
strrchr function 23, 226-227
strspn function 23, 226-227
strtol function 22, 218, 229, 230, 231
sync function 20-21, 130
system function 19-21, 72, 131-135
	an 23, 236
tanh function 23, 223
tdelete function 23, 238-239
tmpnam function 23, 31, 234-235
tfind function 23, 238-239
time function 20, 128, 129, 136-137, 161-163,
times function 20, 137, 157, 235
tmpfile function 23, 195, 233, 235
tmpnam function 23, 29, 31, 195, 233, 234, 235
toascii function 23, 158
tolower function 22-23, 158
toupper function 22-23, 158
trig routines 22-23, 236-237
tsearch function 23, 156, 186, 189, 238-239
ttyname function 22-23, 160, 242
twalk function 23, 238-239
tzset function 23, 161-162
ulimit function 20, 38, 72, 86, 135, 138, 148-150
umask function 20, 68, 72, 86, 106-107, 111, 135, 139
umount function 20, 108, 140
uname function 20, 29, 141
ungetc function 23, 31, 89, 212-213, 243, 303
unlink function 20, 97, 127-128, 142, 233, 235
ustat function 20, 127, 143
utime function 20, 127-128, 137, 144, 302
V

vfprintf function 23, 31, 244-245
vprintf function 23, 31, 244-245
vsprintf function 23, 31, 244-245
W

wait function 20-21, 38, 73, 86, 113, 115, 123-124, 131, 135, 137, 146-147, 157, 277, 279, 302
write function 20-21, 27-29, 68, 74, 80, 101, 110-112, 115, 123, 127, 148-150, 303
Y

y0 function 23, 153
y1 function 23, 153
yn function 23, 153
_exit function 6, 20-21, 32, 58, 72, 73, 74, 115, 123, 131-132, 135, 146-147, 206, 212, 258, 262-263, 275, 278, 285, 287, 293, 299
_tolower function 22-23, 158
_toupper function 22-23, 158
U

Page 320 System V Interface Definition