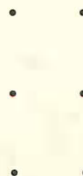


• • •
• •
•

SCO® UNIX®

Development System

Encyclopedia



SCO® UNIX®
Development System
Encyclopedia



© 1983-1992 The Santa Cruz Operation, Inc.
© 1980-1992 Microsoft Corporation.
© 1989-1992 UNIX System Laboratories, Inc.
All Rights Reserved.

No part of this publication may be reproduced, transmitted, stored in a retrieval system, nor translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of the copyright owner, The Santa Cruz Operation, Inc., 400 Encinal, Santa Cruz, California, 95060, U.S.A. Copyright infringement is a serious matter under the United States and foreign Copyright Laws.

The copyrighted software that accompanies this manual is licensed to the End User only for use in strict accordance with the End User License Agreement, which should be read carefully before commencing use of the software. Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc.

The following legend applies to all contracts and subcontracts governed by the Rights in Technical Data and Computer Software Clause of the United States Department of Defense Federal Acquisition Regulations Supplement:

RESTRICTED RIGHTS LEGEND: USE, DUPLICATION, OR DISCLOSURE BY THE UNITED STATES GOVERNMENT IS SUBJECT TO RESTRICTIONS AS SET FORTH IN SUBPARAGRAPH (c) (1) (ii) OF THE RIGHTS IN TECHNICAL DATA AND COMPUTER SOFTWARE CLAUSE AT DFARS 52.227-7013. "CONTRACTOR/SUPPLIER" IS THE SANTA CRUZ OPERATION, INC. 400 ENCINAL STREET, SANTA CRUZ, CALIFORNIA 95060, U.S.A.

Microsoft, MS-DOS, and XENIX are registered trademarks of Microsoft Corporation.
DEC, PDP, VAX, and VT100 are trademarks of Digital Equipment Corporation.
Intel is a registered trademark of Intel Corporation.
UNIX is a registered trademark of UNIX System Laboratories, Inc.

Date: 10 December 1991
Document Version: 3.2.4D

Preface **1**

Documentation conventions	2
Development System documentation set	2

A **7**

acct.h header file	7
agent.h header file	7
ANSI	7
ANSI C	8
a.out file	8
a.out.h header file	8
aouthdr.h header file	8
Application program	9
Archive library	9
Argument	10
ar.h header file	13
ASCII	13
Assembler	15
Assembly language	15
assert.h header file	15
audit.h header file	15

B **17**

BASIC	17
bcopy and bzero routines	17
Binaries	17
Binary Coded Decimal (BCD)	18
Block device driver	19
bool.h header file	19
Bourne shell, sh	20
BSIZE	20
buf structure	21
Buffer	21
Buffer manipulation routines	22
Byte	22
Byte order	23

C language compatibility with assembler	25
C manifest constants	26
C preprocessor	27
C programming language	27
C shell, csh	28
C++	28
C2 security	29
Canonical processing	29
Canonical queue	30
Character routines	31
Character device driver	32
Child processes	32
Clock, system	34
COBOL	34
Command	35
Common Object File Format (COFF)	35
Context switch	40
Control characters	41
Core file	42
core.h header file	43
Crypt libraries	43
cscope	43
ctype.h header file	43
curses	44
curses.h header file	46

D

47

DARPA	47
Data conversion routines	47
Data symbol	48
Database Management Systems (DBMS)	48
Database manipulation routines	48
Dates	49
dbd (disk block descriptor)	50
dbm.h header file	51
Debugging	51
Delimiter	55
Device drivers	56
Device number	63
Device special files	63
dial.h header file	64
Directory	65
Directory operation routines	66
dirent.h header file	67
dir.h header file	67
DMA (Direct Memory Access)	67
Domain	68
DOS and OS/2 cross-development environment	69
DSSSL (Document Style Semantics and Specification Language)	70
dumprestor.h header file	71
Dynamic library	71

E

73

Encryption	73
environ variable	81
Environment	81
eqn	82
errno.h header file	83
Error handling	83
eti.h header file	87
Effective UID (EUID) and Effective GID (EGID)	87
Executable file	87
exit	88
Extended terminal interface (ETI)	88

fatal.h header file	89
fcntl.h header file	89
Field	89
FIFOs (named pipes)	89
File	90
File access system calls	92
File and record locking	93
File descriptor	94
File structure	95
File table, system	96
filehdr.h header file	97
Filename	97
Filesystem access tables	98
Filesystems	100
Filter	101
float.h header file	102
Flow control	102
Flush	103
form.h header file	105
FORTTRAN	105
FSS (File System Switch)	105
ftw.h header file	105
Function (C language)	106
Function prototypes	107
Function (shell script)	109

grap	111
Graphics adapters	111
group file	114
grp.h header file	114

H 115

Hashing	115
Header files	115
High-level language	116
Host adapter	117
HyTime	117

I 119

I18N	119
ieee754.h header file	119
i_errno.h header file	119
Ilist	119
INGRES	119
Inode	120
Inode table, system (in-core)	121
Intel binary compatibility specification (iBCS)	122
Internationalization	123
Internet addresses	124
Interpreted language	125
Interrupt	126
Intrinsics	126
Inumber	126
I/O subsystem	127
ioctl routines	127
IPC facilities	138
ISAM files	143
isam.h header file	145
ISO (International Organization for Standardization)	145

J 147

Job Control	147
-------------------	-----

K **149**

Kernel	149
Korn shell, ksh	149

L **151**

LANG environment variable	151
langinfo.h header file	152
Latin alphabet No. 1	153
ldfcn.h header file	155
lex	156
Lexical analysis	156
Library routines	157
limits.h header file	159
Line discipline	159
linenum.h file header	160
Link editor	160
Link kit	161
lint	162
listen.h header file	162
Locale	162
locale.h header file	165
Localization	165
lock.h header file	166
Locking files and records	166

M **167**

m4	167
Macro	167
Magic number	169
Major and minor numbers	169
make	170
malloc	170
malloc.h header file	171
Manual pages	171
Math routines	174
math.h header file	176

m_defs.h header file	176
mdevice file	176
Memory allocation routines	177
Memory management	178
Memory models	179
memory.h header file	180
menu.h header file	181
Message catalog	181
Messages queues (IPC)	182
Messages (STREAMS)	184
MMDF	184
mnttab.h header file	184
mon.h header file	184
Motif	185
Motif window manager (mwm)	185
mouse.h header file	185
Multibyte characters	186

N **187**

nan.h header file	187
nlist.h header file	187
NLSPATH environment variable	187
nl_types.h header file	188
Null pointer	188

O **189**

Object code	189
OMF (Object Module Format)	189
Open Desktop, SCO	190
Open Software Foundation (OSF)	190
Optimization	190
Option	192
OS/2 cross development environment	192
OSF/Motif	192
OSI (Open Systems Interconnection)	193

Page	197
Page faults	197
Paging	198
panel.h header file	199
Parent process ID (PPID)	199
Parsing	199
passwd file	200
Pathname	200
Permissions, file access	201
Perms list	204
Permuted index	209
pic	209
Pipes	209
poll.h header file	211
Portable programs	211
POSIX standard	212
Preprocessor	213
proc structure	213
Process	213
Process control routines	217
Process group ID	219
Process group leader	219
Process ID (PID)	219
Process subsystem	219
prof.h header file	219
Profilers	220
Program	220
prot.h header file	220
prototypes.h header file	220
Pseudo terminals (pty's)	221
pwd.h header file	221

qsort	223
queue_t data structure	223
Quit signal	223

Random-number generation routines	225
Raw I/O	226
Real user ID and real group ID	226
Record	226
Record locking	226
regex.h header file	227
Region data structure	227
Regular expression	227
reloc.h header file	230

Scancodes	231
SCCS (Source Code Control System)	232
sccs.h header file	233
Scheduling	233
scnhdr.h header file	233
SCSI (Small Computer Systems Interface)	234
sdevice file	235
search.h header file	236
Section	236
Security	236
security.h header file	237
Semaphores	238
Semaphores (System V)	238
Semaphores (XENIX)	241
setjmp.h header file	241
SGML (Standard Generalized Markup Language)	242
sgtty.h header file	242
Shared libraries	243
Shared memory	245
Shared memory, System V	246
Shared memory, XENIX	248
Shells	249
Signals	249
Sockets	253
Source code	254
spl (set priority level)	255

Stack	256
Standard I/O	256
Standards	257
Static linking	257
stdlib.h header file	258
Sticky bit	258
storclass.h header file	258
STREAMS Interface	259
Streams (Standard I/O)	260
String manipulation	261
string.h header file	262
stropts.h header file	263
Structure	263
Subsystems	264
Super block	264
Super user	266
Swap space	266
Swapping	267
Symbol table	268
Symbolic links	268
syms.h header file	268
Syntax	269
System calls	269
System V Interface Definition (SVID)	269

T

271

Terminal Access Method (TAM) transition library	271
tam.h header file	271
Target machine	271
tar.h header file	271
tbl	272
tcap.h header file	272
TCB (Trusted Computing Base)	272
TCP/IP	272
termcap	273
terminfo	274
termio.h and termios.h header files	275
t_errno variable	275
Time	276

time.h header file	277
timezone variable	277
tinfo.h header file	277
TLI (Transport Layer Interface)	277
Translation Lookaside Buffers (TLB)	279
troff	279
Trusted facilities	280
tty structure	283
Type size summary	285
types.h header file	286
TZ environment variable	286

U

289

UART (Universal Asynchronous Receiver/Transmitter)	289
unistd.h header file	289
UNIX Operating System	289
User address space	290
User ID (UID)	290
user structure	291
/usr/group	291
ustat.h header file	291
Utility	291
utime.h header file	292
utmp.h header file	292

V

293

values.h header file	293
varargs.h header file	293
Variable	293
vhand	296
Virtual address mapping	296

W **297**

White space	297
Widget	297
Window (curses)	298
Word	298

X **299**

X Window System	299
xdata.h header file	300
XENIX cross development	300
XENIX operating system	304
X/Open portability guide (XPG)	304
x.out.h header file	305
XTI (X/Open Transport Interface)	305
xti.h header file	309

Y **311**

yacc	311
------------	-----

Z **313**

Zombie state	313
--------------------	-----

0-9 **315**

80x86-family processors	315
-------------------------------	-----

Bibliography **317**

Preface

The SCO UNIX System V/386 Development System *Encyclopedia* contains brief, alphabetically arranged articles on a variety of topics. Our goal is to provide guide-type information in a reference format, supplementing the manual pages with some of the descriptive information required to understand how to develop applications on SCO UNIX System V/386. Types of information to be found include:

- system concepts and related background information.
- summary articles on topics such as "Standard I/O" and "Math routines" that group the Section S system services by topic and reference other documents that users may consult for additional information.
- miscellaneous information that answers common questions our users have had in the past

Articles that begin with a number, such as 8086-family processors, are located in the section 0-9 at the end of the alphabetized section. These *Encyclopedia* articles refer to manual pages, the SCO UNIX Development System Documentation Set, and other books and articles that are available commercially. For more information on obtaining these references, see the Bibliography in this encyclopedia.

NOTE The "(q.v.)"¹ string is the convention used in this *Encyclopedia* to mark terms that are used in one article but have their own article. The term being referenced by the "(q.v.)" is in italics. For example, the article "File descriptor table, user area" includes the following:

Each entry is a *file descriptor* (q.v.) and corresponds to an open file.

This indicates that there is an article entitled "File descriptor" that contains additional information of interest.

1. q.v. stands for *quod vide* and means "see that" as in to see that which is referenced.

Documentation conventions

SCO documents use font changes and other typographic conventions to distinguish text elements. The following table shows these conventions:

Font conventions

Example	Entity
cc or cc(CP)	command. The "CP" indicates the manual page section in which the command is documented.
<i>libc.a</i>	filename
action	placeholder
⟨Esc⟩	keyname
open or open(S)	system calls, library routines, kernel functions, C keywords. The "S" indicates the manual page section in which the command is documented.
buf	structure
b.errno	structure member
\$HOME	environment or shell variable
SIGHUP	named constant
login	output
"clobber"	jargon
"adm3a"	data value
date	user input

Development System documentation set

The contents of the SCO UNIX System V/386 Development System documentation set are illustrated here. In addition to these books, one set of *Release and Installation Notes* is shipped with each copy of the Development System software.

Overview	<i>Developer's Overview</i>	
Reference	<i>Programmer's Reference Manual (2 volumes)</i>	<i>Encyclopedia</i>
Guides	<i>Programming Tools Guide</i>	<i>Developer's Topics</i>
	<i>Debugging Tools Guide</i>	<i>User Interfaces Guide</i>
Optional	<i>Device Driver Writer's Guide</i>	<i>Macro Assembler Writer's Guide</i>

The books included with the Development System are described here.

Developer's Overview

introduces the Development System facilities and gives general information about developing software to run on SCO UNIX systems and the supported cross-development environments

Programmer's Reference Manual

a two-volume set that includes manual pages for the entire Development System, including sections (CP), (DOS), (FP), (S), and (XNX). See the "Manual pages" article in the *Encyclopedia* for a description of these manual page sections.

Encyclopedia

contains articles that give background information about system internals, descriptions of facilities, and other general issues. Articles are arranged alphabetically.

Developer's Topics

a collection of technical papers about topics of interest to users of the Development System. Many of these papers include extensive examples that illustrate features that SCO added to the porting base.

Programming Tools Guide

provides generally useful information about programming tools and their use in developing software. It also provides implementation-specific details about the ANSI-conforming C compiler that is provided with the Development System.

Debugging Tools Guide

provides generally useful information about the debugging tools, and their use in tracking down and eradicating problems in C and assembly language programs. In addition, it includes information on using dbXtra to debug C and C++ programs in a Motif-based windowing environment.

User Interfaces Guide

introduces the facilities that are available for developing user interfaces, and gives detailed instructions for using `curses(S)`, the Extended Terminal Interface (ETI), and writing user interfaces that can be run on ASCII terminals. This book includes numerous examples of `curses` programs.

In addition, the following books are available for Development System customers who need them:

Device Driver Writer's Guide

Provides guide and reference information describing how to write device drivers for SCO UNIX systems. This book is sold separately.

Macro Assembler Writer's Guide

Provides information on the assembly language for the Intel 286, 386, and 486 processors. Developers who are writing device drivers or applications that have tight performance requirements may want to write portions in assembly language. Return the coupon enclosed with the Development System documentation to receive a copy.

Using the Documentation

We recommend the following approach to using this documentation set. In general, we suggest the following:

1. All users should read the *Developer's Overview* thoroughly, soon after installing the software. Experienced users may skim some sections, but this book gives a comprehensive overview of the Development System's capabilities.
2. Use the *Encyclopedia* to look up topics about which you need more information. The *Encyclopedia* gives both summary and background information on many topics, and includes cross-references to sources of more in-depth information.
3. The *Programmer's Reference Manual* contains complete reference information about all capabilities provided in the Development System. New users are warned, however, that the manual pages provide reference material that may be difficult to utilize if they do not understand basic concepts.

The *Programmer's Reference Manual* contains separate sections for Development System commands (CP), system services (S), Development System file formats (FP), and other topics. The pages in each section of the *Programmer's Reference Manual* are arranged alphabetically, although some related facilities are grouped together on a page. If you cannot locate the correct manual page, use the permuted index to find the one that contains the information you want. The "Manual pages" article in the *Encyclopedia* explains how to use the permuted index.

4. The *Programming Tools Guide* and *Debugging Tools Guide* are mainly for users who have not used one of these tools before, or who want to reacquaint themselves with some aspect of that tool's operation. These books are good places to look when the *Encyclopedia* and *Programmer's Reference Manual* are giving you lots of detail, but not enough guidance. Nonetheless, even experienced programmers will discover helpful hints and practical tips on the tools discussed in these books.

Commercial books and articles

A number of fine books and articles are published commercially that discuss how to develop software on the UNIX Operating System. We have not attempted to replicate guide information on all of these topics; manual pages are provided, and many articles in the *Encyclopedia* list additional sources of information on their subject matter. In addition, the *Encyclopedia* includes a large “Bibliography” that will interest users of the Development System.

In particular, we recommend that all developers have the following standard textbooks on their shelves:

Brian Kernighan and Dennis Ritchie, *The C Programming Language*, 2nd Edition.

Marc Rochkind, *Advanced UNIX Programming*.

W. Richard Stevens, *UNIX Network Programming*.

Maurice Bach, *Design of the UNIX Operating System*

See the “Bibliography” in the *Encyclopedia* for full citations if you are not familiar with these books.

A

acct.h header file

The `<sys/acct.h>` header file specifies constants and structures used for system accounting.

See also “Header files” in this *Encyclopedia*.

agent.h header file

The `<agent.h>` header file specifies constants and structures used to perform windowing on a class of graphics terminals invented at AT&T Bell Laboratories and sold by various vendors.

See also “Header files” in this *Encyclopedia*.

ANSI

The American National Standards Institute (ANSI) is a private body that establishes standards in the computing industry. The membership of ANSI includes both commercial organizations and individuals. Typically, ANSI standards in the computing industry are developed by committees drawn from the full range of members, while final decision-making is done by a group of larger vendors, who must achieve “substantial agreement” on the material being considered.

ANSI standards do not have any intrinsic authority; products may be manufactured which do not meet a particular standard, even if the manufacturer has previously approved that standard. ANSI standards, however, are well-respected and may become *de facto* industry standards, effectively requiring compliance from manufacturers. As well, compliance with a standard may become a legal requirement for a product to be considered for government-related projects.

ANSI standards have been established for a number of programming languages, including Ada, C, and FORTRAN. The C compiler provided with the SCO UNIX System V/386 Release 4.0 Development System conforms to the ANSI C (q.v.) standard. The compiler’s conformance with this standard ensures a high degree of portability for code that is written within the parameters set out by the standard.

ANSI C

The *ANSI* (q.v.) C specification, currently ANSI X3.159-1989, defines a common C syntax plus the synopsis of a basic set of library routines.

The `cc -ansi` compilation command provides a fully conformant ANSI C compiler. The `__STDC__` definition in a header file indicates that the code supports the function prototype feature of the ANSI specification. The SCO UNIX operating system compiler defines `__STDC__` as 0 (zero) when ANSI features are not desired, and as 1 when the `cc -ansi` option is specified.

See the *Programming Tools Guide* for implementation defined characteristics for the C compiler supplied with the Development System software.

a.out file

a.out is the default name of the executable object created by the *link editor* (q.v.). The term is often used to reference the executable object, even if it is assigned a different name with the `-o` option to the `cc(CP)` or `ld(CP)` *command* (q.v.).

On SCO UNIX systems, *a.out* files use AT&T's COFF (*Common Object File Format*) (q.v.). The format for *a.out* files is defined in the `<a.out.h>` file. On XENIX systems or when the `-xenix` option is specified to the `cc(CP)` command, *a.out* files use the format defined in the `<x.out.h>` file. The format of the relocatable object file that corresponds to the *x.out* executable file format is commonly referred to as OMF (*Object Module Format*) (q.v.).

a.out.h header file

The `<a.out.h>` header file specifies constants and structures that describe the format for COFF (*Common Object File Format*) (q.v.) binaries.

See also "Header files" in this *Encyclopedia*.

aouthdr.h header file

The `<aouthdr.h>` header file specifies a structure used by the *a.out* (q.v.) header found in the optional header section of COFF (*Common Object File Format*) (q.v.) executables.

See also "Header files" in this *Encyclopedia*.

Application program

An application program is a user-level *program* (q.v.) in a system. Such programs are usually unique to one type of business or activity, although some application programs can be used in a variety of business or task situations. An accounting application, for example, might be applicable to many different businesses.

Application programs execute in *user address space* (q.v.) and are written with *system calls* (q.v.) and *library routines* (q.v.) that are documented in the System Services (S) section of the manual pages. Some applications may require specialized devices that are implemented with *device drivers* (q.v.) that execute as part of the *kernel* (q.v.). For instance, a medical or scientific application may include drivers for microscopes or other equipment; a robotics application may include drivers for process I/O boards or sensors.

See the *Developer's Overview* for information about designing and coding application programs.

Archive library

An archive *file* (q.v.) or archive library is a collection of files, most commonly, object modules. The command `ar`(CP) is used to build and maintain archive libraries.

Note the following about archive files:

- Each of the files within an archive is called a member.
- An archive produced by `ar` begins with a *magic number* (q.v.), followed by the members, each preceded by a file header.
- A null byte (ASCII 0 (zero)) is inserted between files if necessary, ensuring that each file begins on a 16-bit word boundary. The size given reflects the actual size of the file, exclusive of padding.
- There is no provision for empty areas in an archive file.

Archive libraries can contain members of any type, however, they are primarily used to hold object files. Routines that are used in several different programs are commonly stored in an archive, as are general utility routines provided by the system. If there are any COFF files in the archive, `ar` will automatically create the first archive member, which contains the symbol table. If there are any files with long filenames (longer than 14 characters) they are stored in the archive using a short filename convention. These shortened filenames are used as an index entry into a map table of long filenames, created and stored in the archive.

When using an archive library to satisfy unresolved external reference from a program, the link editor copies the object modules from the archive library into the appropriate `.text` and `.data` sections in the program's object file. This has the following ramifications:

- When a routine in an archive library is changed, all programs that call that routine must be relinked to capture the new version of the routine. This contrasts with *shared libraries* (q.v.), whose routines are loaded when the program starts executing, so that new versions of the routine are implemented without recompiling and relinking the code that calls the routine.
- Each program that calls a routine from an archive library, gets its own copy of the routine's code. This results in duplication of the same code on the disk and in memory when the executable files are run as processes.

NOTE The duplication of the same code in memory occurs only if the processes are executing different files.

Consider the following distinctions when deciding whether to use a non-shared or a shared library:

- When archive libraries are used the program's private copy of the library routine must be loaded into memory at execution time, along with the rest of the program. So it may take longer to load the program than if a shared library were used.
- Executable code is usually larger (and hence occupies more disk space for storage and more memory for execution) when archive libraries are used rather than shared libraries.
- Archive libraries are supported on virtually all UNIX operating systems.
- Archive libraries are generally easier to build and maintain than are shared libraries.

Argument

An argument is additional information that is passed to a command or function. Both command line arguments and function arguments are described in the sections below.

Command line arguments

A command line argument is a character string or number that follows the command name and is separated from it by *white space* (q.v.). The shell passes command line arguments to the invoked program.

Most programs classify their arguments as either:

- option** controls the execution or output of the command. Command line options are usually immediately preceded by a minus sign (-). Some commands require that options be identified by other characters, a plus sign (+) for instance. Other commands, such as **tar(C)**, do not require that options be identified by a special character at all. Finally, some options can themselves take arguments.
- operand** specifies files or directories affected by the command or other information needed by the command.

The following command line illustrates:

```
pr -t -h Heading file
```

All elements after the **pr** are arguments. **-t** and **-h** are options; **Heading** is an argument to the **-h** option; and **file** is an operand, in this case the name of the file to be printed.

When entering arguments on a command line for an application, ensure that each argument is separated from the others by one or more white-space characters. If an argument must contain white-space characters, enclose that argument in double quotation marks, single quotes, or any other escape character provided by the shell. Also enclose in double quotation marks any argument that contains characters recognized by the shell (for example, **<**, **>**, **|**, and **^**).

Conventionally, when you enter a command line, the *shell* (q.v.) counts the arguments, stores them in memory, then executes the specified program. The program can then access both the count and the values of the arguments.

For a C program to access the arguments, the *main function* (q.v.) must have two parameters: *argc*, an integer variable containing the argument count, and *argv*, an array of pointers to the argument values. When a C program begins execution, *argc* contains the count, and each element in *argv* contains a pointer to one argument.

The *Bourne shell* (q.v.) and *Korn shell* (q.v.) access command line arguments through the shell *variables* (q.v.) **\$0**, **\$1**, ..., **\$9**. While the number of arguments a shell script can have is large, only the first ten are directly available as variables. Initially, **\$0** contains the name of the script. **\$1** is the first argument, **\$2** is the second argument, and so on. The **shift** command (see **sh(C)** or **ksh(C)**) discards **\$1**, renames **\$2** as **\$1**, **\$3** as **\$2**, and so on, making the tenth argument **\$9**. In this way, all of the arguments are available to shell scripts.

The **getopts(C)** command can be used to parse shell script command line arguments that follow certain rules. See the man page for more information. The following example parses the **-a**, **-b**, and **-c** arguments.

```
1      :
2      while getopts abc: argument
3      do
4          case $argument in
5              a | b)    FLAG=$argument;;
6              c)        OPTION=$OPTARG;;
7              ?)        echo "
8      usage: cmd -a -b -c \"value\"
9              cmd -ab -c \"value\"
10     ";exit;;
11         esac
12     done
13     shift `expr $OPTIND - 1`
```

- line 1 ensure that the script is executed under Bourne shell
- line 2 the **while** loop continues as long as **getopts** returns values for the option choices of **a**, **b**, or **c**. As each argument is parsed, it is assigned to the **argument** variable.
- lines 4 to 7 test the value of the **argument** variable. If it is set to "a" or "b," the value of the **argument** variable is assigned to **FLAG** in line 5. If the **argument** variable is "c," the contents of the value contained in the **OPTARG** variable set automatically by **getopts** is assigned to **OPTION** in line 6.
- lines 7 to 10 display usage message if no options are supplied
- line 14 shift the pointer to the current argument by the number of options processed by **getopts**. **OPTIND** is set by **getopts** to contain an index to the next argument to be processed.

Function arguments

In the C language, arguments to *functions* (q.v.) are enclosed within a pair of parentheses immediately following the function name. The number of arguments can be zero or more; if there is more than one argument, they must be separated by commas. The System Services (S) section of the *Programmer's Reference Manual* provides formal definitions of the functions (system calls and library routines) provided with the Development System software. Each definition describes the number and data type of the argument(s) expected by the function.

ar.h header file

The `<ar.h>` header file specifies constants, macros and structures used when creating an archive file or *archive library* (q.v.).

See also "Header file" in this *Encyclopedia* and the manual page for `ar(CP)`.

ASCII

ASCII, the American Standard Code for Information Interchange, is an ANSI (q.v.) standard that defines a character set for compatibility between data services. ASCII is used by UNIX and many other operating systems, as well as a majority of data display and print devices. It is the single most widely known and used character code set in the computer and communications industries. The ASCII character set is the native character set used by SCO UNIX System V and XENIX operating systems.

ASCII uses a *binary* (q.v.) encoding scheme that is seven bits wide (plus parity), to represent printable characters and non-printable device and communication *control characters* (q.v.). The code set includes 128 codes which represent 33 control characters, the 26 upper- and 26 lowercase letters of the english alphabet, 10 numerals, and 33 other printing characters. ASCII has changed over time, however, and some old equipment may not support certain characters (lower-case characters were not defined until 1965) and may change the value of other characters (@ and <Esc>, for example). Equipment that does not support the current ASCII character set is considered to be obsolete.

It is useful to note that ISO 646 also defines a seven bit character encoding scheme that matches ASCII in all respects except for the names associated with control characters. ISO 646, which is also known as the International Reference Version (IRV) is the starting point for all international standards that define character sets (ISO 4873 and ISO 6937 for example).

Octal values of ASCII character set															
Control characters				Printing characters											
000	nul	020	dle	040	sp	060	0	100	@	120	P	140	`	160	p
001	soh	021	dc1	041	!	061	1	101	A	121	Q	141	a	161	q
002	stx	022	dc2	042	"	062	2	102	B	122	R	142	b	162	r
003	etx	023	dc3	043	#	063	3	103	C	123	S	143	c	163	s
004	eot	024	dc4	044	\$	064	4	104	D	124	T	144	d	164	t
005	enq	025	nak	045	%	065	5	105	E	125	U	145	e	165	u
006	ack	026	syn	046	&	066	6	106	F	126	V	146	f	166	v
007	bel	027	etb	047	'	067	7	107	G	127	W	147	g	167	w
010	bs	030	can	050	(	070	8	110	H	130	X	150	h	170	x
011	ht	031	em	051	)	071	9	111	I	131	Y	151	i	171	y
012	nl	032	sub	052	*	072	:	112	J	132	Z	152	j	172	z
013	vt	033	esc	053	+	073	;	113	K	133	[	153	k	173	{
014	np	034	fs	054	,	074	<	114	L	134	\	154	l	174	
015	cr	035	gs	055	-	075	=	115	M	135	]	155	m	175	}
016	so	036	rs	056	.	076	>	116	N	136	^	156	n	176	~
017	si	037	us	057	/	077	?	117	O	137	_	157	o	177	del

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

It is fairly common practice among system vendors and other computer professionals to refer to alternate code sets which use a full eight bits for data encoding as "Eight-bit ASCII". While this is a misnomer, it is important to know that there are several such code sets which have been put forward over the years. The SCO International Development System uses the ISO 8859/1 8-bit character set, also known as *Latin alphabet No. 1* (q.v.), which includes characters from most Western European languages.

Assembler

An assembler is a translating program that accepts instructions written in the *assembly language* (q.v.) of the computer hardware and translates them into the binary representation of machine instructions. In many cases, the assembly language instructions map 1-to-1 with the binary machine instructions.

The **masm**(CP) command invokes the assembler. The syntax of the assembler used for the Intel 8086-family processors (q.v.) is described in the *Macro Assembler Writer's Guide* that can be ordered from The Santa Cruz Operation.

Assembly language

Assembly language is a programming language that uses the instruction set that applies to a particular microprocessor. MASM is an assembly language for the Intel 80286, 80386, and 80486 microcomputers; it is documented in the *Macro Assembler Writer's Guide*.

A C language program can call an assembly language routine, and an assembly language program can call C language routines. When compiling C code that calls or will be called by assembly language code, you must use one of the **-M** options to the **cc** command to specify the calling convention: **-M0**, **-M1**, and **-M2** establish the 8086/80286 calling conventions; **-M3**, the default, establishes the 80386 calling conventions.

See also "C language compatability with assembler" in the *Developer's Topics*, and "Assembler" in this *Encyclopedia*.

assert.h header file

The `<assert.h>` and `<xenix/assert.h>` header files specify a routine used to abort a program written in the C *programming language* (q.v.) if a specified condition is false.

See also "Header file" in this *Encyclopedia* and the manual page for **assert(S)**.

audit.h header file

The `<sys/audit.h>` header file defines constants and structures used for the audit device discussed on the *audit(HW)* manual page. It should be included in the code for any program implemented as part of a *trusted facility* (q.v.), after the `<sys/unistd.h>` header file.

B

BASIC

BASIC (Beginner's All-purpose Symbolic Instruction Code) is a high-level conversational programming language with simplified I/O instructions, string handling, and debugging. BASIC is often implemented as an interpreter rather than a compiler, meaning that the *source code* (q.v.) is translated into machine language each time the program is executed rather than being stored in machine language. Because of its ease of use, BASIC can be used for prototyping programming concepts and creating user interfaces quickly.

bcopy and bzero routines

These two routines are provided when you have *TCP/IP* (q.v.) installed and you compile using the `cc -lsocket` option. The `bcopy` routine is used to move blocks of bytes in memory from one location to another. This routine calls `memcpy(S)`; however, the order of arguments is different between the two routines.

The BSD version of `bcopy` handles overlapping source and destination addresses, but `memcpy` does not and can produce unexpected results.

The `bzero` routine sets a block of bytes to zero. This routine is similar to `memset(S)`; `bzero(addr, bytes)` does the same as `memset(addr, 0, bytes)`.

Binaries

A binary, or object module, is a *program* (q.v.) that is executable by the operating system. Binaries are created by `ld(CP)`, the *link editor* (q.v.), and under SCO UNIX systems are in the *COFF* (*Common Object File Format*) (q.v.), unless they are compiled with the `-xenix` option, in which case they are in *x.out* format.

Most binaries are upwardly compatible but not backward compatible within the Intel chip family. For example, an 8086 binary can run on an 8086, 80286, 80386, and 80486 processor, but an 80286 binary cannot run on an 8086 processor. Also, with the SCO development system, you can cross-develop binaries. For example, on an 80386 processor you can generate a binary for an 80286 or an 8086 chip.

See the article "80x86-family processors" for more information on processors.

Binary Coded Decimal (BCD)

Binary coded decimals (BCD) is an encoding scheme for decimal numbers in which each digit of a decimal number is represented by its 4 bit binary value. There are two types of BCD numbers: packed and unpacked.

Unpacked BCD numbers are stored one digit to a *byte* (q.v.), with the value in the lower four bits. In assembly language, unpacked BCD variables are defined with the **DB** directive. For example:

```
unpackedr      DB      9,2,5,2,8,5,1
```

This variable occupies 7 bytes whose contents look like:

byte 7	9	00001001
byte 6	2	00000010
byte 5	5	00000101
byte 4	2	00000010
byte 3	8	00001000
byte 2	5	00000101
byte 1	1	00000001

Packed BCD numbers are stored two digits to a byte, with one digit in the lower four bits and one in the upper four bits. In assembly language, packed BCD variables are defined with the **DT** directive. For example:

```
packed DT      9252851
```

This variable occupies 4 bytes whose contents look like:

byte 4	9	00001001
byte 3	2,5	00100101
byte 2	2,8	00101000
byte 1	5,1	01010001

Whether least-significant digits can come first or last depends on how you write the calculation routines that handle the numbers. Calculations with BCD numbers are discussed in the *Macro Assembler Writer's Guide*. SCO does not provide C support for BCD; and the Development System does not provide library routines for BCD arithmetic.

Block device driver

A block device driver transfers data to and from a device in units of fixed-sized blocks, usually corresponding to *buffers* (q.v.) in the system buffer cache. Devices, such as disk drives, that are controlled by a block driver have a “b” in the first character of the *modes* field when you issue an `ls -l` command for the *device special file* (q.v.). A block device driver may also support a character interface, which is called a raw interface.

Data transferred between the *kernel* (q.v.) and the device resides in *BSIZE* (q.v.) blocks of memory that are managed by buffer headers, an instance of the *buf structure* (q.v.). The buffers are typically managed by the kernel in a buffer cache, used to reduce disk accesses and thus improve system performance.

Block device drivers typically include the following entry point routines:

Entry point routines for block drivers

Routine	Purpose
xxinit	initialize the device when the kernel is first booted
xxopen	open device and perform error and protection checking
xxstrategy	queue an I/O request
xxbreakup	size data into units appropriate for the particular controller
xxstart	interact directly with devices I/O ports to check whether device is ready to accept another transfer request and, when ready, to start the transfer
xxprint	log a message when an error occurs during a block transfer
xxintr	respond to interrupts issued by the device, including interrupts that notify the driver that a transfer is complete
xxclose	disable interrupts, clear device registers, eject media, and do other cleanup operations that are appropriate on the last close of a device.

See the *Device Driver Writer's Guide* for more information about how block drivers are coded.

bool.h header file

The `<bool.h>` header file specifies constants used for boolean operations.

See also “Header files” in this *Encyclopedia*.

Bourne shell, *sh*

The Bourne shell was created by S. R. Bourne at Bell Laboratories in the late 1970s. The Bourne shell reads lines typed at a terminal and interprets them as commands for execution. The shell provides a user interface to the operating system and operates on either commands entered at the command line or on those supplied in a file. The commands supplied in a file are called a shell script. The Bourne shell provides a series of tools and commands, described on the *sh*(C) manual page, that range from descriptions of filename wildcards and redirect symbols to programming constructs that can rival the complexity of C programming language applications. The Bourne shell provides the function execution capabilities of *C shell*, *cs**h* (q.v.) and *Korn shell*, *ks**h* (q.v.), but has neither of the command history features of these shells.

A good description of the Bourne shell is provided in *A Practical Guide to UNIX System V* by Mark G. Sobell, which is described in the Bibliography in this *Encyclopedia*.

Bourne shell scripts

A shell script consists of a series of Bourne shell commands in a file; a script can be a simple listing of commands, such as the following example that displays the date, who is logged on, and the current path:

```
date
who
pwd
```

A shell script can also be an application with programming tasks such as prompts, file access, and recursion. A common use for Bourne shell scripts is each user's *.profile* file. Other common uses for shell scripts are for system administration and for programming activities when a program needs to be constructed quickly and execution speed is not a factor. Examples of more complex shell scripts are described in the "Argument" and "Function" articles in this *Encyclopedia*.

BSIZE

The `<sys/fs/s5param.h>` file defines the **BSIZE** manifest constant as representing the number of *bytes* (q.v.) in each *buffer* (q.v.) in the system buffer cache, as well as the file system block size. Application programs that optimize performance by bundling data for the most efficient disk access should use this manifest constant to define the size of the transfer rather than hard-coding the buffer size into the program. This increases the code's portability, because the value is automatically adjusted when the code is compiled on a system that uses a different buffer size.

buf structure

The `buf` structure is defined in the `<sys/buf.h>` header file. This *structure* (q.v.) is used by a *block device driver* (q.v.) to describe buffers for data blocks. These blocks are typically managed by the kernel to form a buffer cache. Some *character device drivers* (q.v.) use the `buf` structure as a header for data blocks that are not part of the system buffer cache.

The `buf` structure contains the following information:

- Buffer status to identify if the buffer is in use or busy, whether a read or write activity is in progress, and whether an error has occurred
- *Device number* (q.v.) to identify the device associated with the I/O operation
- Link list information to identify where the data is in the linked list
- *Byte* (q.v.) count of the data to be transferred
- Virtual address of the *buffer* (q.v.)
- Error codes connected with this activity
- I/O starting time
- In some cases, the `buf` structure also contains user process information that connects an instance of the *proc structure* (q.v.) to the data being transferred.

In the `buf` structure, the `b_flags`, `b_error`, `b_resid`, and `b_start` *fields* (q.v.) can be set by a *device driver* (q.v.), all other fields are read-only. The `b_flags` field must never be cleared by a *device driver* (q.v.).

Buffer

A buffer is a storage space in computer memory where data is stored temporarily.

See also “Standard I/O” in this *Encyclopedia*.

Buffer manipulation routines

The following buffer-manipulation routines are useful for working with areas of memory. These routines always take a length or count *argument* (q.v.).

Routines for memory manipulation

Routine	Description
<code>memcpy(s1, s2, c, n)</code>	copies characters from buffer <i>s2</i> to <i>s1</i> , stopping after the first occurrence of character <i>c</i> or after <i>n</i> characters have been copied
<code>memchr(s, c, n)</code>	returns pointer to first occurrence of character <i>c</i> in the first <i>n</i> characters of buffer <i>s</i>
<code>memcmp(s1, s2, n)</code>	compares the first <i>n</i> characters of buffer <i>s1</i> to buffer <i>s2</i> ; return value indicates whether <i>s1</i> is lexicographically less than, equal to, or greater than <i>s1</i>
<code>memcpy(s1, s2, n)</code>	copies <i>n</i> characters from <i>s2</i> to <i>s1</i>
<code>memset(s, c, n)</code>	sets first <i>n</i> characters in buffer <i>s</i> to the value of character <i>c</i>

Include the `<string.h>` header file (q.v.) for compliance with XPG3. For compliance with AT&T SVID Issue 2 and earlier releases of X/Open, include the `<memory.h>` header file (q.v.). The memory manipulation routines are discussed on the **memory(S)** manual page (q.v.).

Byte

A byte is a unit of storage in computer memory. On SCO UNIX System V and the XENIX system a byte is 8 bits (binary digits), the equivalent of one character of text.

Byte order

Byte order is the order in which data is stored in computer memory. The following table summarizes the byte ordering for **short** and **long** data types for various hardware platforms. Because byte-order may be different on different computers, any program dependent on the byte order is nonportable.

Byte ordering summary

CPU	Byte order †	
	short	long
8086*	b0 b1	b0 b1 b2 b3
80286	b0 b1	b0 b1 b2 b3
80386	b0 b1	b0 b1 b2 b3
80486	b0 b1	b0 b1 b2 b3
PDP-11®	b0 b1	b2 b3 b0 b1
VAX-11®	b0 b1	b0 b1 b2 b3
MC68000	b1 b0	b3 b2 b1 b0
AT&T 3b2	b1 b0	b3 b2 b1 b0
Sun SPARC RISC	b1 b0	b3 b2 b1 b0

† The least significant byte ("lsb") of the data item is b0; b1 is the next least significant, and so on.

* This ordering applies to the Microsoft C compiler for the 8086.

Byte order is commonly described as *little endian*, *big endian* or *bi-endian*. In the little endian model, the least significant byte is at the lowest address in memory; that is, the information that is displayed first when read from left to right on the screen. In the big endian model, the most significant byte is at the lowest address in memory. Bi-endian means that both models are supported.

The following code example illustrates how to determine which model is in use on the system:

Determining the system endian

```

1      #define VAL 0xaabbccdd
2      main( )
3      {
4          union (
5              struct (
6                  unsigned char  b0, b1, b2, b3;
7              ) byte;
8              int    word;
9          ) offset;
10         offset.word = VAL;
11         printf("0x%x is represented as:\n", VAL);
12         printf("0x%x 0x%x 0x%x 0x%x\n", offset.byte.b0, offset.byte.b1,
13                                     offset.byte.b2, offset.byte.b3);
14     }

```

- line 1 The macro **VAL** is defined to be hexadecimal 0xaabbccdd because this value provides four bytes of values that are easy to read in the output.
- line 4 to 11 The **offset** variable is a union containing the **byte** structure and the **word** integer.
- line 12 The **offset.word** structure field is set to **VAL**.
- lines 13 and 14 The byte positions are displayed.

The output of this program shows which model is supported:

Little endian output	Big endian output
0xaabbccdd is represented as: 0xdd 0xcc 0xbb 0xaa	0xaabbccdd is represented as: 0xaa 0xbb 0xcc 0xdd

C

C language compatibility with assembler

One of the convenient features of the *C programming language* (q.v.) is that it provides a ready interface to assembly language. Therefore, C language programs can call routines written in assembler, and assembly language programs can call C functions, including the many standard C library functions.

Though the exact procedures differ depending on the type of processor being used (80386 or 80286/8086), the general guidelines to follow when writing code that mixes assembler and C are generally the same.

The C language conventions in terms of data types and sizes are used when passing data between C and assembler.

All arguments are passed on the stack. The size, order, and format of the data pushed on the stack are dictated by the processor being used. Return values are stored in various internal registers which hold the data values or are pointers to larger structures; the specific registers used are, again, processor dependent.

In general, the assembler code is responsible for the cleanup or housekeeping considerations. The assembler code should take care to preserve all registers that are not preserved by the calling convention before calling a C function. The assembler code can use the stack for temporary storage but must take care to remove any values (including arguments) from the stack before returning to its caller.

For more information and detailed code examples on this topic see the "C language compatibility with assembly language" chapter in the *Developer's Topics*.

C manifest constants

The following is a list of manifest constants, also called “predefined macros.” The second column of the table lists the names that are non-namespace polluting, that is, they do not violate the rules for naming system macros set out in the ANSI, XPG3, and POSIX standards for C. Most of the constants are for internal use only and should not be used in user applications. The information that follows is useful for understanding header files. The constants are:

Manifest Constant	Nonpolluting Version	Description
i386	_i386	80386 processor
M_BITFIELDS	_M_BITFIELDS	The compiler supports bitfields; not used
M_COFF	_M_COFF	COFF-format file being generated
M_I386	_M_I386	80386 processor
M_I8086	_M_I8086	8086 processor
M_I186	_M_I186	80186 processor
M_I286	_M_I286	80286 processor
M_I86	_M_I86	Generic
M_I86CM	_M_I86CM	Compact model
M_I86HM	_M_I86HM	Huge model
M_I86LM	_M_I86LM	Large model
M_I86MM	_M_I86MM	Medium model
M_I86SM	_M_I86SM	Small model
M_LDATA	_M_LDATA	Large model data
M_LTEXT	_M_LTEXT	Large model text
M_SDATA	_M_SDATA	Small model data
M_STEXT	_M_STEXT	Small model text
M_INTERNAT	_M_INTERNAT	Link with <code>cc -lintl</code>
MSDOS		Create MS-DOS executable
M_SYS3	_M_SYS3	System III compatible; not used
M_SYS5	_M_SYS5	System V compatible; not used
M_SYSIII	_M_SYSIII	System III compatible; not used
M_SYSV	_M_SYSV	System V compatible; not used
M_UNIX	_M_UNIX	SCO UNIX
M_WORDSWAP	_M_WORDSWAP	Swapped words. Old versions of XENIX, for the 8086, had longs with the words swapped
M_XENIX	_M_XENIX	SCO XENIX
M_XOUT		Create XENIX executable
OS2		Create OS/2 executable
	_POSIX_SOURCE	Strict POSIX conformance
	STDC	=1: conforms to one of the standards (ANSI, POSIX, XPG3) =0: conforms to one of the standards (ANSI, POSIX, XPG3) plus some extensions. (if neither <code>_XOPEN_SOURCE</code> or <code>_POSIX_SOURCE</code> are also defined then implies strict ANSI conformance.)

(Continued on next page)

(Continued)

	<code>_STRICT_NAMES</code>	Do not pollute name space
	<code>_SVID</code>	Strict AT&T SVID Issue 2 conformance
unix	<code>__unix</code>	UNIX operating system
	<code>_XOPEN_SOURCE</code>	Strict XPG3 conformance
	<code>_NO_PROTOTYPE</code>	Prototypes are not available (see <code>rcc</code>)
	<code>_KR</code>	Obey K&R rules
cplusplus	<code>__cplusplus</code>	Defined for the C++ compiler

C preprocessor

The C preprocessor is a program that prepares C source code for the C compiler. Its main functions are to perform file inclusions and macro substitutions, and to provide for conditional compilation. The C preprocessor is called automatically by the `cc(CP)` command; if `rcc(CP)` is being used, the preprocessor `cpp(CP)` can be invoked separately.

The `-P`, `-E`, or `-EP` options to the `cc` command produce a listing of the preprocessor's output. The `-C` option can be used with these options to preserve comments during preprocessing so that they print out in the preprocessor output.

C programming language

C is a general programming language that features economy of expression, control flow, data structures, and a variety of operators. It can be used to perform both high-level and low-level tasks. All UNIX operating systems are written in C.

Two C compilers are provided in this release:

- Microsoft 6.0 C Compiler is accessible with the `cc(CP)` command. This C compiler conforms with ANSI C as described in the standard ANSI X3.159-1989 Programming Language — C.
- A derivative of AT&T's Portable C Compiler, accessible with the `rcc(CP)` command. This C compiler provides a syntax similar to that described in Kernighan and Ritchie's *The C Language* (first edition). It is provided for customers who require a C compiler that adheres to the old syntax, although it might not be supported in future releases.

See the *Programming Tools Guide* and the *Standards Conformance Statements* for implementation-specific information about the C compiler.

C shell, *cs*h

The C shell, *cs*h(C), is an interactive command-line interpreter that serves as an interface between the user and the kernel. The syntax of the C shell is based on the *C programming language* (q.v.), whence it received its name. It is a sophisticated programming environment that supports such features as functions, control flow, logical branching, and input/output procedures. Some special features of the C shell are command aliasing, command history, and built-in integer arithmetic. Some versions of the C shell support *job control* (q.v.); the version supplied with SCO UNIX System V/386 does not.

The C shell is one of three *shells* (q.v.) commonly available on the UNIX operating system; the other two are the *Bourne shell*, *sh* (q.v.) and the *Korn shell*, *ksh* (q.v.). In comparison to the Bourne shell, the C shell is considered more versatile because it allows command history and more advanced array handling capabilities. However, C shell is slower than the Bourne shell. The Korn shell, the most recently developed shell, combines the best features of the other two shells.

See the *cs*h(C) manual page for more detail on the C shell.

C++

C++ is a programming language based on C that supports object-oriented programming. The language is very similar to C, but has a number of extensions. In most cases, though not always, C code will have the same meaning in C++ as it does in C.

The central feature of C++ is the notion of a *class*. Classes are user-defined data types. They are similar to *structures* in C, but, unlike structures, classes allow the user to define functions which operate on the data objects in the class. One kind of function, the *constructor* function, describes how new objects of the class are initialized.

Other features of C++ are inheritance of class attributes by other classes, dynamic typing, and operator overloading.

For more information on C++, refer to *The Annotated C++ Reference Manual* by M. Ellis and Bjarne Stroustrup, Addison-Wesley, 1990, and to *The C++ Programming Language* by Bjarne Stroustrup, Addison-Wesley, 1986.

C2 security

See "Security."

Canonical processing

Canonical processing means resolving special characters such as backspace or delete before the received data is given to the calling program. By default, canonical processing is enabled automatically for a terminal device. Canonical processing also translates international characters. Canonical processing is not done when:

- The terminal controller handles its own character interpretation. Conventional canonical processing creates considerable overhead on the system. In the last few years, terminal controller manufacturers have eliminated this overhead by putting the canonical processing capability into the board itself. This permits terminals to operate at vastly faster speeds than was permitted with having the central processing unit perform the canonical processing.
- When the `VMIN` or `VTIME` flags are enabled. These flags specify that a `read(S)` call not be satisfied until a minimum number of characters are entered (`VMIN`), or until a specified time interval elapses (`VTIME`). Refer to `termio(M)` for more information on these flags.
- When the data is accepted directly by the program and processed by its own methods.

The `termio(M)` manual page describes guidelines that are used when canonical processing takes place. The `termios(M)` manual page describes the POSIX variation for canonical processing. Refer to the `c_cc` array description on the `termio(M)` manual page for information about which characters are resolved by canonical processing.

Canonical processing is enabled when the `tty` device driver constant `ICANON` is enabled; this constant is described on the `termio(M)` manual page. The default condition is that `ICANON` is enabled automatically.

The following example shows how the `ICANON` constant can be disabled from a user program:

Disabling canonical processing

```

1      #include <stdio.h>
2      #include <termio.h>
3      #include <fcntl.h>
4
5      main()
6      {
7          int fd;
8          struct termio tio;
9
10         fd = open( "/dev/tty", O_RDWR );
11         ioctl( fd, TCGETA, &tio );
12         tio.c_lflag &= ~ICANON;
13         ioctl( fd, TCSETA, &tio );
14     }
```

lines 1 to 3 Header files are included for use with the program. The *stdio.h* file contains the routines used for file I/O. The *termio.h* file defines the `ioctl` and assignment constants. The *fcntl.h* file defines the read-write flag `O_RDWR` for the `open` call.

line 7 An instance of the `termio` structure is defined. Refer to the `termio(M)` manual page for more information on the fields in this structure.

line 8 An arbitrary tty device file is opened for reading and writing.

line 9 Using the `ioctl(S)` system call, the `termio` structure for the tty device is brought into memory to populate the `termio` structure instance `tio`.

line 10 The *line discipline* (q.v.) control field, `c_lflag`, is ANDed with “not” `ICANON` to remove `ICANON` thus disabling canonical processing.

line 11 Finally, the `termio` structure is written back to the tty device.

Canonical queue

When a process writes data to a terminal device, the user context (task-time) portion of a driver puts the data into an output queue, and the interrupt routine transfers it from the queue to the device. However, when a process reads data, the situation is slightly more complex. This is because the operating system provides for some processing of characters on input, at the option of the requesting process.

Some special characters, such as `<Bksp>`, depend on surrounding characters and must be treated in context. To handle this, drivers may use *canonical processing* (q.v.) to ensure that characters are processed on input.

When the *interrupt* (q.v.) routine receives data, it places the data in the raw queue. At task time, the driver decides how much processing to do. The user process has the option of requesting raw input, in which it receives data directly from the raw queue.

Cooked input is the opposite of raw input. Cooked input is data that has been processed for ERASE, line kill, DELETE, and other special treatment. The task-time device driver function `canon(K)` transfers data from the raw queue to the canonical queue. This function performs backspace and line-kill functions, according to the options set by the process using the `termio(M)` flags.

Character routines

The character routines let you classify and convert individual characters in a variety of ways, including conversion between uppercase and lowercase characters. The routines operate on the ASCII character set only. The character routines are presented in the following table.

Character routines

Routine	Description
<code>isalnum(c)</code>	tests <i>c</i> for alphanumeric character (letters and digits)
<code>isalpha(c)</code>	tests <i>c</i> for alphabetic character (uppercase and lowercase letters)
<code>isascii(c)</code>	tests <i>c</i> for ASCII character (0-127)
<code>iscntrl(c)</code>	tests <i>c</i> for control character (ASCII 0-31 and 127)
<code>isdigit(c)</code>	tests <i>c</i> for decimal digit (0-9)
<code>isgraph(c)</code>	tests <i>c</i> for printable character except space (ASCII 33-126)
<code>islower(c)</code>	tests <i>c</i> for lowercase character
<code>isprint(c)</code>	tests <i>c</i> for printable character (ASCII 32-126)
<code>ispunct(c)</code>	tests <i>c</i> for punctuation character (neither control nor alphanumeric)
<code>isspace(c)</code>	tests <i>c</i> for whitespace character (space, tab, carriage return, newline, vertical tab, or form feed)
<code>isupper(c)</code>	tests <i>c</i> for uppercase character
<code>isxdigit(c)</code>	tests <i>c</i> for hexadecimal digit (0-9, a-f, A-F)
<code>toascii(c)</code>	converts <i>c</i> to ASCII code
<code>todigit(c)</code>	converts <i>c</i> to a digit (0-9)
<code>toint(c)</code>	converts <i>c</i> to an integer
<code>tolower(c)</code>	tests <i>c</i> and converts to lowercase if uppercase
<code>_tolower(i)</code>	converts <i>i</i> to lowercase (unconditional)
<code>toupper(c)</code>	tests <i>c</i> and converts to uppercase if lowercase
<code>_toupper(i)</code>	converts <i>i</i> to uppercase (unconditional)

For more information about the character classification and conversion routines, see the `ctype(S)` and `toascii(S)` pages in the *Programmer's Reference Manual*.

Character device driver

A character device driver transfers information from the user program to the device; it may or may not do an intermediary transfer to the kernel. Devices, such as terminals and specialized devices as well as the character-based disk and tape devices, that are controlled by a character driver have a “c” in the first character of the `modes` field when you issue an `ls -l` command for the special device file.

Character drivers are used for devices such as multiport boards, scientific equipment, and other equipment, where the data is transferred with any scheme that the driver defines, rather than being confined to the use of the buffer cache as are *block device drivers* (q.v.).

Character drivers typically include the following entry point routines:

Routine	Purpose
<code>xxinit</code>	initialize the device when the kernel is first booted
<code>xxopen</code>	open device and perform error and protection checking
<code>xxread</code>	transfers data to the user's address space
<code>xxwrite</code>	copies data from user's address space
<code>xxhalt</code>	set or clear device registers when the system is shut down so that devices are ready for initialization after a warm boot
<code>xpoll</code>	called to reprime device at every clock interrupt
<code>xxproc</code>	provides hardware-dependent code in a line discipline
<code>xxioctl</code>	perform hardware-dependent functions appropriate for the device
<code>xxintr</code>	respond to interrupts issued by the device, including interrupts that notify the driver that a transfer is complete
<code>xxclose</code>	disable interrupts, clear device registers, eject media, and do other cleanup operations that are appropriate on the last close of a device.

See the *Device Driver Writer's Guide* for more information about how character device drivers are coded.

Child processes

A child process is a *process* (q.v.) that is spawned by another process, called its parent process. A child process inherits some of the parent's process environment, but once spawned exists as a separate process with its own environment. When you look at a child process with the `ps -ef` command, the child's PPID (parent process identification) matches the PID (process identification) of the parent process.

Forking child processes

The `fork(S)` system call creates a child process, which is effectively a duplicate of the parent process that created it. This call is implemented using a copy-on-write scheme, in which data pages are copied only when a write operation is requested, to avoid unnecessary copying. `fork` duplicates the parent's data and stack segments, which are the memory segments for processes, plus its environment to the child. The child shares its parent's text (code) segment, which conserves memory space.¹ This scheme enables child processes to initialize more quickly than if they had to recreate the entire environment rather than merely modify parts of the inherited environment.

`fork` returns different values to the parent and child. The parent receives the child's PID, which can never be 0, and the child receives 0. The program uses these return values to determine which is the child process and which the parent process (because they are both executing the same program) and takes different branches in each process' code.

After checking the value returned by `fork`, the parent and child execute different branches of the same program in parallel. If the child needs to execute a different program, it issues an `exec(S)` call. `exec` replaces the text and data segments of its caller with those of a new program read from a file specified in the call. `exec` does not alter its caller's environment; a child process may be executing a different program, but it still has access to the file descriptors of files opened by the parent. Those files, however, might have been modified by the child between the `fork` and the `exec`.

A child issues an `exit` system call to terminate normally; this call takes a parameter whose value is returned to the child's parent. A child may also be terminated abnormally by a signal issued by the kernel, a user, or another process. When a child terminates, either normally or abnormally, the operating system sends a `SIGCLD` signal to the parent process.

Waiting for a child process to terminate

The parent process can continue execution in parallel with the child process or it can wait for a child process to terminate. If the parent needs to wait for the child to terminate it can issue either the `wait(S)` or the `waitpid(S)` system call.

If the parent needs to wait for any child process to terminate it can simply issue the `wait()` system call. If the parent needs to wait for a specific child process to terminate the `waitpid()` system call can be used. `waitpid()` allows the actual process ID of the child, returned by the `fork()` system call, to be specified as an argument. Thus the parent process can spawn several children and specify the one for which it is waiting. If `waitpid()` is given the value -1 for the process ID argument it behaves identically to the `wait()` system call.

1. By default, all programs are reentrant; that is, by default all processes have an unmodifiable, and therefore, sharable, text segment.

Whether or not the parent waits for the child to terminate, it receives a SIGCLD signal when the child terminates.² The parent process can catch this signal, then issue a `wait` to learn what happened to the child process.

Clock, system

The system motherboard on computers using the AT, EISA, and MCA buses have a programmable interval timer with several channels, one of which is used as the system clock. This channel is set so it generates interrupts at 100 Hz, so that a clock tick is generated every 1/100 second.³ This conforms to the *Intel binary compatibility specification (iBCS)* (q.v.). Applications that need to determine the frequency of the system clock in ticks per second can access the `HZ` environment variable or use the `gethz(S)` or `sysconf(S)` system calls.

The clock interrupt handler is activated on every clock tick and maintains the per-process user and system times as well as the current time of day. It also provides a triggering mechanism for profiling and the *device driver* (q.v.) `timeout(K)` kernel function.

NOTE The system time can become inaccurate if the `spl(K)` kernel routines in a device driver are used improperly. Refer to the *Device Driver Writer's Guide* for more information.

COBOL

COBOL, COmmon Business Oriented Language, is a high-level programming language designed for business and commercial applications.

-
2. By default, the SIGCLD signal is ignored. The parent process can be set up to receive this signal. This is explained on the `signal(S)` manual page and in the "Interprocess Communications" chapter in Marc Rochkind's *Advanced UNIX Programming*.
 3. Earlier releases of the 3.2 operating system used a 60 Hz clock.

Command

A command is an instruction that a user types at a computer terminal keyboard. It can be the name of an executable file or a shell script that can be processed or executed by the computer.

The Operating and Development System commands are documented in the following sections of the *User's Reference*, *Programmer's Reference Manual* and the *Administrator's Reference* :

C	commands that are included with the operating system and can be executed by any user who has execute permissions for the command and the appropriate access permissions to other system resources.
ADM	commands that are included with the operating system that require super user permissions. Most (ADM) commands are intended for system administrators.
CP	commands that are packaged with the Development System and can be executed by any user with appropriate permissions.
XNX	programming command that can be used only for XENIX cross-development.

Command line

A command line consists of the command name followed by any *arguments* (q.v.) required by the command or optionally included by the user. The manual page for a command includes a command line synopsis that shows the correct way to issue the command, and descriptive text that explains the use of options and arguments that are supported.

Common Object File Format (COFF)

COFF is the AT&T format for object modules, libraries, and executable binaries for use with the UNIX operating system. Both COFF and *Object Module Format* (OMF) (q.v.) format are fully supported by the software generation and maintenance tools and have full runtime support in the UNIX kernel.

The COFF format has the following limitations:

- COFF binaries cannot be executed on XENIX systems before XENIX System V release 2.3.
- COFF binaries cannot be executed on XENIX systems that use the 286 architecture.

The COFF file organization and associated include files are shown in the following diagram:

HEADER DATA	INCLUDE FILES
File Header	filehdr.h
Auxiliary Header Information	aouthdr.h
.text section header	scrhdr.h
.data section header	scrhdr.h
.bss section header	scrhdr.h
RAW DATA	
.text section data	
.data section data	
RELOCATION DATA	
.text section relocation data	reloc.h
.data section relocation data	reloc.h
LINE NUMBER DATA for debuggers	
.text section line numbers	linenum.h
.data section line numbers	linenum.h
SYMBOL TABLE	
.text, .data, and .bss section symbols	syms.h storclass.h

COFF File Organization

The Microsoft C compiler can generate either COFF or OMF/x.out binaries depending on command line options. Additional tools that provide common functionality for both formats such as `ar(CP)` and `ld(CP)` utilize a front-end that executes the appropriate binary-specific tools relative to the file types provided as command line arguments. All other development system utilities are specific to either COFF or OMF/x.out file manipulation; these utilities include most standard AT&T tools and those for complete XENIX cross-development. In general, these tools cannot be used in a symmetrical fashion.

Most libraries are distributed in both COFF and OMF versions so that system calls and library services are readily available for linking into either binary type. Note, however, that non-386 COFF binaries are not supported by either the development tools or the kernel. The full range of OMF/x.out non-386 binaries are still fully supported for backward compatibility with older SCO products.

Choosing between COFF and OMF/x.out

Developers should select the desired binary format for objects, archives, and executables before compiling any source code. Both binary formats have advantages and disadvantages; the choice should be made considering factors such as:

- format of existing objects and libraries
- whether you want to use shared libraries
- debugging services you want to use during development
- size of the executable objects
- target environments to which you need portability

Developers who are making libraries available as part of their products should provide both OMF and COFF objects and archives so that the customer can use either binary format. In this case, we recommend that each archive be produced with a separate compilation from source code. We do not recommend that you convert objects and libraries using the conversion tools after compilation; this method cannot be used for code that uses shared libraries or file locking as explained in the section “General incompatibilities between COFF and OMF/x.out”.

Conversion tools

All versions of the UNIX operating system Development System use two translation or conversion tools to support COFF object files:

- | | |
|----------------|--|
| cvtomf | converts OMF relocatable object files produced by pass three of the Microsoft compiler to COFF relocatable object files. cvtomf may be considered an optional fourth pass of the compiler used for COFF generation. It fully translates OMF object files to COFF binaries; when used with the -g option, it provides symbolic debug data. |
| cvtcoff | converts COFF relocatable object files, archives, and executables to the OMF/x.out format. Symbolic debug data are provided only for fully linked executables. cvtcoff was used under SCO XENIX System V release 2.3 (under the name coffconv) to support the linkage of COFF objects and archives into x.out binaries. Under the SCO UNIX System V/386 Release 3.2 Operating System its use is largely restricted to translating COFF symbolic debug data into x.out format for the Codeview debugger. It is also used by the Microsoft compiler to translate COFF modules produced by the AT&T assembler (as) when an x.out binary is requested on the command line. |

These conversion tools are used by specific Development System tools, and although they can be used manually, this is not generally recommended.

General incompatibilities between COFF and OMF/x.out

There are two areas of fundamental incompatibility between COFF and x.out files:

- | | |
|------------------|---|
| shared libraries | Only COFF files are capable of using shared libraries. Shared libraries are themselves COFF format files, and only COFF executables provide a well-defined mechanism for requesting the use of one or more shared libraries at load time. COFF executables that use shared libraries do not translate to x.out executables properly; <i>cvtcoff</i> explicitly warns the user of this incompatibility. It is unlikely that x.out binaries will ever be augmented to support shared libraries. |
| file locking | <p>The two binary formats receive different kernel handling for file locking. COFF binaries use advisory locking by default to preserve compatibility with other systems. Under releases of the XENIX operating system, before 2.3, all file locking was mandatory, whereas releases of UNIX System V, before 2.1, provided only advisory locking. XENIX 2.3 was the first time that COFF runtime support was incorporated into the SCO kernel.</p> <p>COFF binaries do not provide the mechanism for requesting either type of lock at load time. This issue affects conversions because code whose logic depends on one type of file locking may not be appropriate under the other type of locking. The result is that conversions between COFF and OMF/x.out format may change the program's locking behavior, with unexpected results.</p> |

Incompatibilities between COFF and OMF/x.out binaries

COFF and OMF/x.out binaries may have different link or runtime characteristics, even though the source code used to produce the binary is identical. This is the result of two historical facts:

- kernel differences between SCO XENIX and traditional AT&T kernels
- differences between SCO XENIX and AT&T library interfaces

To support both COFF and OMF/x.out binaries, the SCO kernel makes some distinctions between the binary types at runtime and may function differently in some cases. The most notable case concerns file locking. The SCO UNIX kernel traditionally uses advisory locking with COFF binaries, whereas the XENIX kernel traditionally uses mandatory locking with OMF/x.out binaries.⁴ The SCO UNIX kernel, however, also provides mandatory locking. This is used if a binary has been specifically linked to use mandatory locking or if the SCO UNIX kernel determines from the format that the binary is a XENIX

4. Refer to the "XENIX cross development" article in this *Encyclopedia* for more information on XENIX locking.

binary. In addition, system call parameter values for `lockf(S)` and `fcntl(S)` are different in some cases between SCO XENIX and AT&T System V. Consequently, the kernel interprets these parameters differently, depending on the binary type. This separate interpretation preserves functionality and provides binary support for both COFF and x.out executables. In addition to kernel services, OMF and COFF libraries handle structure layout (size and alignment) and parameter values differently.

The standard header files in the UNIX operating system *include* and *include/sys* directories have been merged and provide code conditional on the binary format being produced, so most of these differences are transparent to the developer. The Microsoft compiler provides manifest constants (`M_XOUT` or `M_COFF`) to signal the binary format being produced. These constants direct header files to provide the correct code in cases where COFF and OMF/x.out libraries and/or kernel services differ. Consequently, structures, function parameter values, and variable types are automatically compatible with library functions and system calls particular to the binary type.

This scheme creates a serious problem if a developer attempts to translate the binary format after compilation. Objects and libraries built using one format may contain binary values (such as function parameters and structure offsets) specific to that binary format. After they are run through a conversion tool, however, these files are linked with incompatible libraries. Moreover, at run time, the kernel makes functional decisions based on the final binary type, not the original binary type that produced the code.

An example of this problem again comes from file locking. The value for read lock requests is different for the COFF and OMF/x.out formats. An object that uses this value and is compiled as COFF contains the COFF-specific value. This is then translated to OMF and linked. At runtime, the kernel interprets the lock request from an OMF/x.out perspective, and creates the wrong lock type.

The files listed below contain `#ifdef` clauses that use `M_COFF` and/or `M_XOUT`. Any source code that uses these files or the related system calls or library routines should never be converted after compilation.

<code>a.out.h</code>	COFF and x.out binary formats
<code>core.h</code>	core file analysis
<code>lockcmn.h</code>	file lock requests for x.out binaries
<code>mon.h</code>	monitor(S) and profiling data ("mon.out")
<code>tinfo.h</code>	terminfo and curses definitions
<code>curses.h</code>	includes <i>tinfo.h</i>
<code>sys/fcntl.h</code>	fcntl(S) and lockf(S) file lock requests
<code>sys/locking.h</code>	locking(S) system call
<code>sys/shm.h</code>	AT&T shared memory segments (IPC)
<code>sys/signal.h</code>	signals; see SIGPOLL in Intro(S) .
<code>sys/utsname.h</code>	uname(S) system call

AT&T portable C compiler (rcc)

Merged header files provide a solution to many binary incompatibilities between the COFF and OMF/x.out binaries. The success of this scheme is dependent on several *manifest constants* (q.v.) provided by the Microsoft compiler and used during preprocessing. It is important to realize that the AT&T compiler rcc(CP) does not automatically provide the same set of constants. The developer must provide these constants manually if any standard header files are used in source modules that are compiled with rcc. Use the `cc -d` command to produce a list of these constants.

Backward compatibility

The 3.2 compiler provides command line options for selecting different levels of backwards compatibility. This enables the compiler to support XENIX cross-development for both UNIX system and XENIX target environments. The prime issue is to use only those kernel services (*system calls*) (q.v.) that are provided by the target operating system. The OMF libraries are partitioned so that system call extensions that were added with XENIX 2.3 and SCO UNIX System V Release 3.2 may be selectively linked or excluded. COFF development does not provide the same service. COFF binaries that use either the 3.2 system calls listed in */lib/386/Slib3.2.a* or the system calls in */lib/libx.a* do not run properly under XENIX before version 2.3 release 4.

Context switch

A context switch occurs when the kernel transfers control of the CPU from the process that is executing to another process that is ready to execute. This happens for the following reasons:

- a user process's time slice expires
- a user process makes a system call that cannot be completed immediately; the process blocks (or "sleeps"), and another process can execute while that process waits to proceed
- after servicing an interrupt, the kernel finds that the currently executing process no longer has the highest priority of runnable processes.

The operating system saves the process's "context" so that it can be restored when it resumes execution after a subsequent context switch. See Maurice Bach's *The Design of the UNIX Operating System* for details about how the system implements a context switch.

Control characters

Control characters are non-printable characters that are typically used for communication and device control, as format effectors, and as information separators. A control character is represented by a code that can be generated by most standard keyboards and by many software applications. Many control characters can be entered only by holding down the <Ctrl> key while pressing another key. Some control characters have dedicated keys assigned to them, for example the <Bksp>, <Return>, <Tab>, <Del>, and <Esc> keys send a control character. Control characters are interpreted by the system's input processing routines.

The ANSI standard for the *ASCII* (q.v.) character set defines a control character as a character whose occurrence in a particular context initiates, modifies or stops an action that affects the recording, processing, transmission, or interpretation of data. A software application is free to interpret them for any purpose desired.

The following table presents a listing of the ASCII control characters that includes: its <Ctrl> key-sequence; its official name; its code value in decimal, hexadecimal, and octal; and its official description.



Sequence	Name	Decimal	Hex	Octal	Meaning
<Ctrl> @	nul	00	00	00	null character
<Ctrl> a	soh	01	01	01	start of heading
<Ctrl> b	stx	02	02	02	start of text
<Ctrl> c	etx	03	03	03	end of text
<Ctrl> d	eot	04	04	04	end of transmission
<Ctrl> e	enq	05	05	05	enquiry
<Ctrl> f	ack	06	06	06	acknowledge
<Ctrl> g	bel	07	07	07	bell
<Ctrl> h	bs	08	08	10	backspace
<Ctrl> i	ht	09	09	11	tab
<Ctrl> j	nl	10	0A	12	line feed
<Ctrl> k	vt	11	0B	13	vertical tabulation
<Ctrl> l	np	12	0C	14	form feed
<Ctrl> m	cr	13	0D	15	carriage return
<Ctrl> n	so	14	0E	16	shift OUT
<Ctrl> o	si	15	0F	17	shift IN
<Ctrl> p	dle	16	10	20	data link escape
<Ctrl> q	dc1	17	11	21	device control 1
<Ctrl> r	dc2	18	12	22	device control 2
<Ctrl> s	dc3	19	13	23	device control 3
<Ctrl> t	dc4	20	14	24	device control 4
<Ctrl> u	nak	21	15	25	negative acknowledgement
<Ctrl> v	syn	22	16	26	synchronous/idle
<Ctrl> w	etb	23	17	27	end transmission block
<Ctrl> x	can	24	18	30	cancel
<Ctrl> y	em	25	19	31	end of medium
<Ctrl> z	sub	26	1A	32	substitute
<Ctrl> [	esc	27	1B	33	escape
<Ctrl> \	fs	28	1C	34	file separator
<Ctrl>]	gs	29	1D	35	group separator
<Ctrl> ^	rs	30	1E	36	record separator
<Ctrl> _	us	31	1F	37	unit separator
<Del>	del	127	7F	177	delete

Core file

A core file contains the image of a terminated *process* (q.v.) that is saved when certain types of errors occur and can be used for *debugging* (q.v.). A core file can also be created by calling the `abort(S)` routine in a program. The output file is always named *core* and is stored in the current directory of the process. A *core* file contains the contents of the CPU registers, stack, and memory areas of the process, at the time it received the *signal* (q.v.) that caused the core dump. For details about the format of a *core* file, see the *core(FP)* manual page. See the *signal(S)* manual page for information about the various signals that can occur.

Debugging a program with a core file

When a program terminates unexpectedly, the *core* file can be examined, using a debugger, to determine the reason for the termination. More information on debuggers and debugging can be found in the *Debugging Tools Guide*.

core.h header file

The `<core.h>` header file specifies constants and macros used to determine the location and size of a *core* file (q.v.).

See also “Header files” in this *Encyclopedia* and the manual page for *core*(F).

Crypt libraries

The “crypt” libraries provide routines for *encryption* (q.v.) of data files to provide an extra measure of security. Domestic customers may request a copy of the crypt libraries from The Santa Cruz Operation Media Department at no extra charge. International customers may request a copy of the crypt libraries that permits only encryption, but not decryption.

See also the “Encryption” article for more information and examples of using the crypt libraries.

cscope

The *cscope*(CP) browser is an interactive program that locates specified parts of code in a set of C source files and gives you the option of editing those files. It can reduce significantly the amount of time you must spend searching for functions, function calls, macros, and variables in the code. Programmers responsible for writing programs (especially large ones) or maintaining existing programs can edit source code more efficiently with *cscope*.

Refer to the *Programming Tools Guide* for more information on *cscope*.

ctype.h header file

The `<ctype.h>` header file specifies constants, macros, variables and routines used to classify *ASCII* (q.v.) coded integer values by doing a table lookup.

See also “Header files” in this *Encyclopedia* and the manual page for *ctype*(S).

curses

curses is a library of C routines that can be used to create character window-based, interactive programs using ASCII and other terminals. It allows programs to be independent of the terminals from which they are run and optimizes the result of any action affecting the display so that the time required to update the screen is minimized.

Windows permit localized portions of the entire screen to be used independently by a program. This allows all program interaction to be organized in an attractive, accessible and relevant manner, unlike with traditional command line interaction. For more information on windows see *windows (curses)* (q.v.).

Terminal independence is achieved through the use of a terminal description database. The terminal description database used by **curses** is either *terminfo* (q.v.) or its predecessor *termcap* (q.v.). These databases contain “capabilities” which describe each different type of terminal. A program written using **curses** must include the *<curses.h>* header file (q.v.), which contains variable declarations for each of the capabilities. At run time, when the program is started, the terminal description database is searched for an entry which matches the current setting of the TERM environment variable. Once found, the values of the capabilities are assigned to the variables and these variables are used by all the **curses** routines to interact with the specific terminal. If the program were run using a different terminal, different set of capabilities would be loaded at run time though the visual effect on the screen would be similar.

There are nearly 200 routines available in the **curses** library although some of them can only be used specifically with either the *terminfo* or the *termcap* terminal description databases. The following are some of the basic **curses** routines which allow a program to manipulate the initial screen before any additional windows have been created:

curses routines

Routine	Description
<code>addch(ch)</code>	writes character <i>ch</i> at current cursor position in window and advances cursor
<code>addstr(str)</code>	writes string <i>str</i> at current cursor position in window and advances cursor
<code>cbreak()</code>	puts terminal in CBREAK mode
<code>clear()</code>	writes blanks to every position in the window
<code>clrtoebot()</code>	clears current line to the right of the cursor and all lines below the cursor
<code>clrtoeol()</code>	clears current line to the right of the cursor

(Continued on next page)

curses routines
(Continued)

Routine	Description
<code>delch()</code>	deletes the character on which the cursor rests
<code>deleteln()</code>	deletes the line on which the cursor rests
<code>echo()</code>	characters that are typed are displayed on the screen
<code>endwin()</code>	used when exiting curses ; resets terminal into proper non-visual mode and moves cursor to lower left hand corner of screen
<code>getch()</code>	reads character from the terminal
<code>getstr(str)</code>	reads string from terminal until a newline is read
<code>getyx(win,y,x)</code>	the current cursor position in the window <i>win</i> is placed in the variables <i>y</i> and <i>x</i>
<code>inch()</code>	gets a character at the current cursor position
<code>initscr()</code>	initializes curses ; determines the terminal type and initializes all curses data structures
<code>insch(ch)</code>	inserts character <i>ch</i> before the character immediately under the cursor
<code>insertln()</code>	inserts a blank line above the line on which the cursor currently rests
<code>move(y,x)</code>	moves cursor to coordinates <i>y, x</i> on screen
<code>mvaddch(y,x,ch)</code>	moves cursor to coordinates <i>y, x</i> on screen; writes character <i>ch</i> at new cursor position
<code>mvaddstr(y,x,str)</code>	moves cursor to coordinates <i>y, x</i> on screen; writes null terminated string <i>str</i> at new cursor position
<code>mvdelch(y,x)</code>	moves cursor to coordinates <i>y, x</i> on screen; deletes the character immediately under the cursor
<code>mvgetch(y,x)</code>	moves cursor to coordinates <i>y, x</i> on screen; reads character from the terminal
<code>mvgetstr(y,x,str)</code>	moves cursor to coordinates <i>y, x</i> on screen; reads string from terminal until a newline is read
<code>mvinch(y,x)</code>	moves cursor to coordinates <i>y, x</i> on screen; gets character at new character position
<code>mvinsch(y,x,ch)</code>	moves cursor to coordinates <i>y, x</i> on screen; insert character <i>ch</i> before character immediately under the cursor
<code>mvprintw(y,x,fmt[,arg...])</code>	moves cursor to coordinates <i>y, x</i> on screen; writes the format string <i>fmt</i> with arguments <i>arg</i> at the new cursor position
<code>mvscanw(y,x,fmt[,arg...])</code>	moves cursor to coordinates <i>y, x</i> on screen; reads characters interpreting them according to the format string <i>fmt</i> and puts them into variables <i>arg</i>
<code>nl()</code>	turns on newline mapping
<code>nocbreak()</code>	takes terminal out of CBREAK mode
<code>noecho()</code>	characters that are typed are not displayed on the screen

(Continued on next page)

curses routines

(Continued)

Routine	Description
<code>nonl()</code>	turns off newline mapping
<code>noraw()</code>	turns off raw mode
<code>printw(fmt[,arg...])</code>	write the format string <i>fmt</i> with arguments <i>arg</i> at the current cursor position
<code>raw()</code>	turns on raw mode
<code>refresh()</code>	updates the terminal screen
<code>scanw(fmt[,arg...])</code>	reads characters interpreting them according to the format string <i>fmt</i> and puts them into variables <i>arg</i>

For information about routines available for multiple windows and additional information about `curses` see the `curses(S)` manual page or the *User Interfaces Guide*.

***curses.h* header file**

The `<curses.h>` header file specifies include statements for the header files that define the routines used to control screen and cursor functions for programs that use the *curses* (q.v.) library.

See also “Header files” in this *Encyclopedia*, the manual page for `curses(S)` and the *User Interfaces Guide* manual.

D

DARPA

DARPA is the Defense Advanced Research Project Agency, that developed the ARPANET networking protocol during the late 1960's and early 1970's, and then the "Internet Protocol Suite" for communications between various Local Area Networks (LAN's). The Internet Protocol Suite is more commonly called the TCP/IP protocol suite.

Data conversion routines

The data-conversion routines convert numbers to strings of ASCII characters and vice versa and convert numbers between various formats. These routines are implemented as functions and are described in section (S) of the *Programmer's Reference Manual*.

Data conversion routines

Routine	Description
a64l(s)	convert base-64 number <i>s</i> to a long
atof(str)	converts <i>str</i> to double-precision number (float)
atoi(str)	converts <i>str</i> to int
atol(str)	converts <i>str</i> to long
ecvt(value, ndigit, decpt, sign)	converts double <i>value</i> to string of <i>ndigit</i> digits and returns a pointer thereto. <i>decpt</i> defines the position of the decimal point; <i>sign</i> is non-null if the result is negative.
fcvt(value, ndigit, decpt, sign)	same as <i>ecvt</i> , but rounds the return value for <i>printf</i> 's %f output of <i>ndigit</i> numbers
gcvt(value, ndigit, buf)	converts double <i>value</i> to null-terminated string in the array pointed to by <i>buf</i>
l3tol(lp, cp, n)	converts a list of <i>n</i> three-byte integers packed into a character string pointed to by <i>cp</i> into a list of long integers pointed to by <i>lp</i>
l3tol(cp, lp, n)	converts list of <i>n</i> long integers <i>lp</i> into a list of 3-byte integers pointed to by <i>cp</i>
l64a(l)	converts long into a base-64 representation

(Continued on next page)

Data conversion routines (Continued)

Routine	Description
sgetl(buffer)	returns a long stored with sputl and places it in memory at the address pointed to by buffer
sputl(value, buffer)	stores a long value in memory starting at address buf
strtod(str, ptr)	returns as a double-precision floating-point number the value represented by str .
strtol(str, ptr, base)	converts string to long integer
strtoul(sptr, dptr, base)	converts string to unsigned long integer with input radix base . sptr points to the beginning of the string; eptr points to the end of the string.

Data symbol

A data symbol in *object code* (q.v.) represents a *variable* (q.v.), which may or may not be initialized. Normally, data symbols reside in read/write memory during execution.

Database Management Systems (DBMS)

Database management systems provide general-purpose management of data for an application. The SCO Open Desktop Development System includes the Embedded Structured Query Language (SQL) database manager. Users can implement alternate database management systems using the facilities of the SCO ISAM files (q.v.).

SQL and ISAM are not supported on XENIX systems. XENIX systems do, however, provide a limited set of *database manipulation routines* (q.v.), documented on the **dbm(S)** manual page, for the simple manipulation of large databases.

Database manipulation routines

The following routines are available when you specify the library **dbm** on the compile line. They allow you the tools to perform simple manipulations of a very large database. For more information, see **dbm(S)** in the *Programmer's Reference Manual*.

NOTE Be sure to use the **-ldb** option when compiling programs using these functions. These functions are only available to programs that use the OMF format.

XENIX database manager routines

Routine	Description
dbmopen(file)	opens a database file for accessing
delete(key)	deletes a key and its associated contents
fetch(key)	accesses data stored under a key
firstkey()	returns the first key in the database
nextkey(key)	returns the next key following any specified key in the database
store(key,content)	stores data under a key

Dates

Dates and *times* (q.v.) on UNIX operating systems vary in terms of their display format and, occasionally, in the way they keep track of time. Internally, the system date is stored in the kernel as the number of seconds since 00:00:00 January 1, 1970 UTC (Universal Coordinated Time, or GMT). This number, referred to as the epoch, is translated to a calendar date using the routines discussed in the “time” encyclopedia article. The system date is initialized when the system is booted, and is used to provide time stamps for file access and modification and by the electronic mail system, as well as dates coded for specific applications.

Date formats

Date and time formats vary from locale to locale. The order in which fields are presented, the delimiters between fields, and even the actual timing scheme (Japanese Emperor or Era time, for example), are subject to *localization* (q.v.). By default, dates are printed using the conventions of the United States of America (the `ctime(S)` and the `asctime(S)` routines only print according to U.S. convention).

It has become increasingly important to design software with *internationalization* (q.v.) and *localization* (q.v.) in mind, so that it makes no assumptions about date and time formats. Such *locale* (q.v.)-independent software should use `strftime(S)` to present dates. This routine gives the programmer complete freedom to display the date in whatever format desired. Furthermore, the `strftime` routine can alternatively print out the date and time in a format according to the local convention for applications that apply localization techniques. The `strftime` routine reads the value of certain locale-related environment variables to obtain information about how to display the date and time. `strftime` is defined to query the `TZ` (q.v.) variable to obtain local time zone information and then check the `LC_TIME` variable to obtain information about how to display the date time.

To see what locales are supported on a particular system, check in the `/usr/lib/lang` directory. To see what locale convention is being currently used, access the `langinfo` database using the `nl_langinfo(S)` routine. If the locale is a supported one, the `langinfo` database stores information about the current locale's date/time formats, names and abbreviations for days of the week

and months of the year, and (if applicable) the Japanese Emperor/Era name and year (see the `nl_langinfo(S)` manual page for more details on this routine and the database).

System commands using datetime formats

The following commands internally call `strftime` in at least one of their options, and so they reflect changes made to either the TZ or the LC_TIME environment variables: `ar(C)`, `cpio(C)`, `lpstat(C)`, `ls(C)`, `mail(C)`, `mailx(C)`, `pr(C)`, `ps(C)`, and `who(C)`.

Calendars

The most commonly used calendar scheme used is the Gregorian calendar. Each calendar year has 365 days, unless it is a leap year, in which case it has 366 days. A year is a leap year if it is evenly divisible by 4 or 400, but not by 100. This “less three days every four hundred years” rule is the difference between the old Julian and modern Gregorian calendars. The conversion in September 1752 resulted in losing the 11 extra days that had built up in the four 400-year cycles since the beginning of the Common Era.

The following code calculates the number of days in a specified Gregorian calendar year:

```
int days_in_year(
    register int yr
) {
    if ((yr % 4) == 0 && (yr % 100) != 0 || (yr % 400) == 0)
        return (366);
    return (365);
}
```

Besides the Julian and Gregorian calendars mentioned above, there are other date schemes that may be used on UNIX operating systems. Most others would be implementation defined; the Japanese Emperor or Era time is usually supported. In this scheme, months and days are the same in the Gregorian calendar, however the Emperor calendar's year is the number of years that the current Japanese emperor has been reigning.

dbd (disk block descriptor)

A *dbd* (disk block descriptor) is a *kernel* (q.v.) data structure that is part of the memory management subsystem, and is used to describe the location of those *pages* (q.v.) of an executing *program* (q.v.) that are not in memory.

There are 4 major types of *dbd*'s.

- The first describes “demand filled” pages. Typically, after a program *exec*'s, only one or two of the program's pages are actually in memory: all the rest are represented by demand fill *dbd*'s that describe the location of the program's text and data in the executable disk file, so that it can be paged into memory when it is referenced.

- A “file backed” `dbd` indicates that while a page is in memory, an exact copy of the page’s contents exists on disk. The memory management system can thus reclaim this physical page very quickly, because the in-memory contents do not have to be written to disk to reuse the page. `dbd`’s that describe a program’s text section are typically never modified, and thus alternate between “demand filled” and “file backed” as they are paged in and discarded.
- Parts of the address space of an executing program must be all zeros when first referenced. This is handled by “demand zero” `dbds`. They indicate that on first reference, a physical page is allocated, zero’ed and returned to the user.
- If the memory management system needs to reclaim a page that has been modified, the contents of that page must be saved first. This is done by writing the page to a swap device. The `swalloc(K)` routine allocates free swap space on a *swap* device, and rewrites the `dbd` for the page in order to indicate the location on *swap* in which the page contents will reside.

Refer to the “Swapping,” “Paging,” “Page,” and “Region” articles in this *Encyclopedia* for more information.

dbm.h header file

The `<dbm.h>` header file specifies routines used to perform simple operations on very large databases.

See also “Header files” and “Database manipulation routines” in this *Encyclopedia* and the manual page for `dbm(S)`.

Debugging

Debugging is the act of analyzing and resolving errors that prevent a program from compiling, or cause execution errors of some sort. Defensive programming helps reduce the need for more rigid error tracking, analysis, and maintenance procedures later in the development cycle. However, because even the most experienced programmers can make a slip of a finger, debugging tools and techniques remain an important component of the software development cycle.

A program that fails to run, or demonstrates consistently undesirable behavior is frustrating, but as long as the behavior continues to be reproducible, it can be analyzed and the source of the problem tracked down. In contrast, a program that fails intermittently is often extremely difficult to debug because the problem lies with inappropriate use of memory or environmental conditions beyond the scope of the software. Solving run time errors requires a greater degree of problem solving, analysis, and experience with debugging tools and techniques.

This article discusses using four major classes of debugging techniques: compiler error messages, `printf(S)` statements placed within the code, creating and analyzing core dumps, and debugging tools.

Compiler error messages

The compiler informs you about errors when compiling. These error messages are often enough for you to identify the problem. Some compiler error messages are fatal and will stop the compilation, whereas other error messages are only “warnings”. Ideally, you should resolve even compiler warning messages; they identify problems that may cause the program to fail when it executes and will certainly cause the program to be non-portable. By carefully reviewing the *syntax* (q.v.) at the offending line and comparing it to the documentation and to code that works correctly, the error can usually be found and corrected.

Debugging with printf(S)

Including `printf(S)` statements in the code is a useful technique for analyzing execution errors. Most programs include `printf` statements that report when a program fails, but these can be supplemented with `printf` statements that are used only for debugging. Strategically placed, a `printf` statement can provide important information about the state of the process, the values of variables, contents of strings, and so on. Assuming that a problem can be localized, this technique can be an efficient way to debug a code segment.

By placing `printf` statements within `#ifdef` blocks, you can enable this debugging mode at compile time by invoking the compiler with the `-D` option. For example, you can use the following code fragment to monitor the value of an integer inside a loop:

```
while (i<j)
{
#ifdef DEBUG
    printf("Loop counter i = %d\n", i);
#endif
    *
    *
    *
}
```

When the program is compiled with `cc -DDEBUG`, the `printf` statement is included and compiled. When the program is run, the `printf` statement reports the value of the counter.

This simple example, while usable, is not sufficient for production code. However, this technique can be extended in a number of ways. For example, by assigning a value with `cc -DDEBUG=value`, and using `#if` in place of `#ifdef`, the previous example could be rewritten to yield more information depending on the *value* assigned:

```
while (i<j)
{
    #if DEBUG==1
        printf("i = %d\n", i);
    #endif

    #if DEBUG==2
        printf("i = %d : j = %d\n", i, j);
    #endif
    .
    .
    .
}
```

The actual expansion of `#if` and `#ifdef` directives is done by the C preprocessor.

Dumping core

Another useful debugging technique is to create and analyze a *core file* (q.v.) that contains an image of the data state of the program. You can create a core file in a number of ways:

- send the **SIGQUIT** *signal* (q.v.) to the program as it is running. Do this by typing the Quit character, typically `<Ctrl>-\\`, at the desired point during the execution of the program.
- add the **abort** function to the *source code* (q.v.) at the point in the program where the state needs to be examined. See the **kill(S)** manual page for more information.
- Use the **kill** command.

Debuggers

Debuggers are tools that allow the user to control the execution of a program to detect logic problems. By using a debugger, a programmer can step through their program one instruction at a time. This allows for each line of code to be tested, ensuring that the appropriate logic is executed. Programs can be run through, single-stepped, or a combination of these effects can be used by setting breakpoints.

Similarities exist among debuggers; most allow for the display of source code and machine instructions and for monitoring the values of *variables* (q.v.). In some instances programs can be patched while being debugged and re-executed. The source information from a debugger can be displayed in either a line-by-line fashion or through a windowed environment. Symbolic debuggers, such as `codeview`(CP), `dbxtra`(CP), and `sdb`(CP), permit the source code for a program to be viewed as the program executes and have the ability to indicate which line of the source code is currently being executed. They also permit stepwise execution of the source code and the ability to examine the current values of variables. Before using a symbolic debugger on a program, compile the program with the `-g` option to `cc` and `rcc` commands, so that the *symbol table* (q.v.) information is available in the executable.

The UNIX System V/386 Development System includes the following debuggers:

- dbxtra** a powerful source level debugger for *C programs* (q.v.), *C++* (q.v.) and *assembly language* (q.v.) programs. There are two versions of **dbxtra**: **dbxtra** is a full-screen windowing debugger available for use on any character based terminal. **dbXtra** is a full-screen windowing debugger for use under *X Window Systems* (q.v.).

dbxtra supports debugging of both C and C++ products in conjunction with SCO C++ products. This debugger can follow the child or parent *process* (q.v.) when a fork is executed. **dbxtra** allows for the suspension of the parent debugger while debugging the child, and returning to the parent session after exiting the child. When using the X Windows based version of **dbXtra**, a programmer can simultaneously debug parent and child processes.
- CodeView** CodeView provides a window-type environment in which to debug C and assembly language programs. In addition to numerous standard debugging functions, CodeView provides the ability to browse source or assembly code, indicating which line will be executed next. **CodeView** can debug both *COFF* (q.v.) and *OMF* (q.v.) *executable files* (q.v.).

sdb	sdb is an interactive command line debugger. Through sdb , a program's execution can be monitored and controlled by entering sdb commands on the command line. sdb provides a means for displaying variables to be accessed symbolically and viewed in the correct format. It also provides a means of viewing the source code a line at a time and the ability to produce a stack trace.
adb	adb is a debugging tool for C and assembly language programs, and is currently the only debugger available for debugging 286 binaries. adb provides a method for viewing non-ASCII data files with line mode commands. adb can single step through a program allowing program patches, at the source level, to be applied to a program.
ctrace	ctrace provides the ability to follow the execution of a C program statement-by-statement. When you execute a program under ctrace , each statement is listed on either the standard output or a file, followed by the name and value of any variable referenced or modified in that statement. Endless loops can be detected through ctrace 's built-in ability to print a warning message after each 1000th iteration of a loop.

For more information on debugging and the debuggers mentioned in this article see *Debugging Tools Guide*.

Delimiter

A delimiter is a character used to group sets of characters as a unit on a command line or to separate fields in ASCII files that contain tables of data. Delimiters are used when *parsing* (q.v.) a *string* (q.v.) of characters, which in this sense means to subdivide a string into component parts.

Command line delimiters

Command line delimiters enable the shell to properly interpret the string of characters typed on the command line. Typically, a shell interprets delimiters on the command line as follows:

- <Space>, <Tab>, or <Return> characters separate the command, options, and arguments from one another
- the semi-colon (;) separates two commands entered on the same line
- the vertical bar or pipe symbol (|) indicates the output from the preceding command serves as input to the following command
- the less-than (<) and greater-than (>) signs redirect *stderr*, *stdin*, and *stdout*, usually to a different file

The following example illustrates the use of command line delimiters:

```
$ date; ls /bin | sort > temp.out(Return)
```

In this example, the string `date` is delimited by the semi-colon command separator, which indicates that the next command is totally separate and could have been entered on a separate line. This string is then interpreted by the shell to be the `date` command. The string `ls` is delimited by a (Space) that separates the command from its argument, `/bin`. The pipe delimiter indicates that the output of `ls /bin` is to serve as input to the `sort` command. (This is not really necessary because the output of `ls` with no options is already sorted alphabetically.) The output redirection symbol, `>`, redirects the output from `sort` from *stdout* to the `temp.out` file.

Field delimiters

Many UNIX applications read data files in which fields within a *record* (q.v.) are separated by delimiters; (Space) and (Tab) are common. In the `/etc/passwd` file, which stores entries for each user that can log onto the system, the delimiter is the colon (:). A typical entry is of the form:

```
root:*:0:1:Superuser:/:/bin/sh
```

Here the delimiter divides each `/etc/passwd` entry into the "loginid", "password", "userid", "groupid", "comment", "home directory" and "shell" fields used by the login program.

Device drivers

Device drivers provide the software interface between a device and the operating system. A device driver is a set of routines that communicates with a hardware device and provides a uniform interface to the operating system kernel. It also serves as a manager of data flow and control between a user program and a peripheral device. User-level programs access devices through a *device special file* (q.v.) that looks like a regular file to the program, but is used by the kernel to identify the driver that controls access to the device.

In addition to hardware device drivers that control peripheral devices, there are purely software drivers (sometimes called "pseudo-drivers") that are a set of routines that provide additional functionality to the kernel. For instance, kernel profiling is implemented as a pseudo-driver, and memory itself can be read and written through the `/dev/kmem` and `/dev/mem` pseudo-devices.

For physical devices, a device driver is the end of the data stream closest to an external interface. The principal functions of a device driver are handling an associated physical device and transforming data and information between the external interface and data stream.

Customers can create and install drivers; the *Device Driver Writer's Guide* provides information about writing and installing drivers on UNIX System V/386.

The device drivers currently available in your UNIX system kernel are documented in section HW of the manual pages and listed in */etc/conf/cf.d/mdevice*. Each *mdevice* line declares a driver; each driver is linked into the kernel according to directions in the files in the */etc/conf/sdevice.d* directory. The format of the list is similar to the following example:

Sample mdevice(F) listing

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
first	I	io	first	0	0	0	1	-1
aud	Iocrwi	iocr	aud	0	21	0	0	-1
cr_uni	-	io	cruni	0	0	1	1	-1
du	-	io	du	0	0	1	1	-1
ecc_	oci	ic	ecc_	0	22	0	1	-1
fp	-	iHor	fp	0	0	0	1	-1
sy	orwi	icor	sy	0	3	1	1	-1
.	.	.	.	.	.	.	.	.

The fields (numbered for reference) indicate

- 1 device name
- 2 function list
- 3 characteristics of the driver
- 4 handler prefix
- 5 block major number
- 6 character major number
- 7 minimum units
- 8 maximum units
- 9 DMA channel

Kernel driver tuning

The default kernel contains a large number of drivers (many of which your particular system will never use) so that it can be installed and operate on a large variety of systems. You can remove the device drivers that your system does not need, to give you a smaller kernel, lower memory requirements, or to free up an interrupt vector.

NOTE Never remove a device driver, unless you are certain your software and hardware do not require it.

To remove a driver you must edit the appropriate file in */etc/conf/sdevice.d* by changing the "Y" entry to "N". Some other adjustments may be required; those are described in the "Key to driver tables" on the next page, and indicated with symbols for each driver or device listed. See the *Device Driver Writers Guide* and the *mdevice(F)* and *sdevice(F)* manual pages for more information about device drivers.

Device driver names

The following tables list many of the device drivers used by the SCO UNIX System V/386 Release 3.2 Operating System. The actual drivers used on your system may vary slightly.

Key to driver tables:

-
- * indicates the drivers are in the kernel by default.
 - † indicates that you must remove the "r" (required) flag from the third column in */etc/conf/cf.d/mdevice*. When you remove the *r* flag and rebuild the the kernel environment, you delete the device node. If you install this device later, add the *r* flag to the *mdevice* file, and recreate the device node by using the **mknod** command. See the *mdevice*(F) man page for more information on characteristics of the driver.
 - ** indicates that you must comment out entries in */etc/conf/cf.d/mscsi* in order to remove the driver. (Use * as the comment character.)
 - § indicates that in order to remove the second WD driver, you must add **int wdlintr** to the */etc/conf/pack.d/wd1/stubs.c* file.
-

The following is a list of display mode devices. Note that you cannot mix VGA, CGA, and EGA cards in one machine.

Display device drivers

mdevice name	Device driver name	Removal key
mono	mono graphics card	*†
cga	IBM color graphics adaptor	*†
ega	Enhanced Graphics Adaptor	*†
vga	VGA driver	*†

The following drivers are dependent on Merge and VP/ix.

Merge and VP/ix device drivers

mdevice name	Device driver name	Removal key
merge	DOS Merge	*
dda	Merge and VP/ix direct device access	
vdsp	Merge display	*
vkbd	Merge keyboard	*
vmouse	Merge mouse	*
vpld	VP/ix line discipline	
vpt	VP/ix tty	
svkbd	Stream based merge keyboard	*
svdsp	Stream based merge display	*
mrnat	Merge - AT	*

The following devices might be required by your VP/ix or Merge software. They are also required by parts of the UNIX System V software.

Miscellaneous System V device drivers (Do not remove)

mdevice name	Device driver name	Removal key
ptem	AT&T STREAMS pseudo-ttys (ioctl)	
ptm	AT&T STREAMS pseudo-ttys (master)	
pts	AT&T stream pseudo-ttys (slave)	
ldterm	STREAMS pseudo-tty line discipline	

You only need the following drivers if you have a SCSI Host Adapter.

SCSI controller device drivers

mdevice name	Device driver name	Removal key
Sdsk	SCSI disk	* †
Srom	SCSI CD Rom	†
Stp	SCSI tape	† *
ad	Adaptec SCSI Host Adapter	* * * †
wdha	Western Digital 7000 SCSI Host Adapter	* †

If you do have a SCSI Host Adapter, you might not need the following drivers.

Western Digital controller device drivers

mdevice name	Device driver name	Removal key
wd	Western Digital Disk Controller	* †
wd0	Western Digital Disk Controller #1	* †
wd1	Western Digital Disk Controller #2	* † §

NOTE If you only have one ESDI or ST506 controller, you can remove the second driver, *wd1*.

The following are TCP/IP and NFS drivers.

TCP/IP and NFS device drivers

mdevice name	Device driver name	Removal key
arp	Address Resolution Protocol	*
arpproc	Address Resolution Protocol Proc	*
cp	TCP/IP Copy Protection	*
tcp	TCP stack	*
nfsd	NFS	*
llcloop	Loopback driver	*
icmp	Internet Control Message Protocol	*
ip	Internet Protocol	*
lockd	NFS File Locking	*
socket	BSD socket support	*
timod	TLI Cooperating Module	*
tirdwr	TLI read/write module	*
vty	TCP/IP pseudo-tty (master)	*
ttyp	TCP/IP pseudo-tty (slave)	*

The following drivers are optional TCP/IP drivers.

Optional TCP/IP device drivers

mdevice name	Device driver name	Removal key
nb	NetBios	

The kernel requires only one of the following drivers per ethernet card.

Ethernet device drivers

mdevice name	Device driver name	Removal key
e3A	3com 501	
e3B	3com 503	
e3C	3com 523 Micro Channel	
slip	SLIP Communications Protocol	
tok	Token Ring	
wdn0	Western Digital Ethernet Card	

If you install a mouse, you need the following drivers.

Mouse device drivers

mdevice name	Device driver name	Removal key
evld	event line discipline	
ev	event driver	

You only need the following drivers if you have installed the specific mouse.

Optional mouse device drivers

mdevice name	Device driver name	Removal key
busmouse	Microsoft bus mouse	
kbmouse	keyboard mouse	

You need the following drivers if you install Xsight.

Required drivers for Xsight installation

mdevice name	Device driver name	Removal key
clone	clone driver	*
sp	stream pipe	
str	STREAMS interface	
mpt	non-STREAMS psuedo tty (master)	*
spt	non-STREAMS psuedo tty (slave)	*
mpm	psuedo mouse (master)	
spm	psuedo mouse (slave)	

If one of the hardware devices listed below is not installed on your system, you might be able to remove the driver.

CD-ROM, disk, tape, and miscellaneous hardware device drivers

mdevice name	Device driver name	Removal key
ct	cartridge tape	* †
weitek	Weitek co-processor	
omti	OMTI disk drive	* †
ft	floppy tape driver (QIC-40 and QIC-80)	
mc	Irwin mini tape	
ir	Irwin tape driver	

By default, two serial I/O ports and two parallel ports are configured in the kernel. If you don't use either of the ports, you may remove them one at a time, or both at the same time.

Serial and parallel port device drivers

mdevice name	Device driver name	Removal key
sio	serial I/O	* †
pa	parallel printer	* †

The following drivers are required for debugging.

Debugging device drivers

mdevice name	Device driver name	Removal key
mskdb	kernel debugger	
scodb	kernel debugger	

The following devices are required for system security.

Security device drivers

mdevice name	Device driver name	Removal key
aud	system auditing	* DO NOT REMOVE
sec	C2 security	* DO NOT REMOVE

The default kernel contains all of the following device drivers. Most of these are important and are used by miscellaneous programs.

Miscellaneous device drivers

mdevice name	Device driver name	Removal key
cdb	MC146818 driver (CMOS clock)	* DO NOT REMOVE
cn	console driver	*
cpyrt	copyright	*
da	VGA, EGA, CGA, and mono device driver	*
db	devbuf driver; support for driver memory allocation	*
dk	disk layout support driver	*
err	error messages	*
fd	floppy disk	*
fp	floating point	*
hd	root disk driver	*
ipc	general interprocess communication	* DO NOT REMOVE
log	system messages logger	*
mm	memory	*
msg	messages	*
nmi	non-maskable interrupt handler	*
prf	profiler	* †
rtc	real time clock	*
sem	AT&T semaphores	*
shm	AT&T shared memory	*
ss	Sun River stub	*
sy	/dev/tty, console	*
vx	virtual 8086	*
xsd	XENIX shared data	*
xsem	XENIX semaphores	*

The following drivers are rarely used and are usually not configured in the kernel.

Device drivers

mdevice name	Device driver name	Removal key
xt	Blit Packet Protocol Driver	
sxt	shell layers	
du	RFS driver	
first	MPX kernel hook, used during MPX initialization	* DO NOT REMOVE
ram	Ram disk†	*

Device number

A device number is a number that identifies a device that has a special device file. The device number consists of two parts, the *major and minor numbers* (q.v.). The major number tells the kernel which device driver handles requests for this device. The minor number can be used by the driver to provide more information about a particular instance of the device that it controls (such as the unit number). For example, all the ports on an eight-port serial card could have the same major device number, but have eight separate minor device numbers.

The minor device number part of the device number usually encodes the unit number. However, a *device driver* (q.v.) can dedicate some of the bits in the minor number to indicate special options, such as to use double density in the case of a floppy disk.

In UNIX System V/386, the device number contains both the major and the minor number and is a short integer of type `dev_t`. Use the `major(K)` and `minor(K)` kernel functions to extract the major and minor device numbers, respectively.

User-level applications can use the `uadmin(S)` routine to return a device's major number.

Device special files

A device special file (sometimes called a device node) is a particular type of file that provides an interface to a device and its corresponding *device driver* (q.v.). Programs can access a device special file with the *file access* (q.v.) system calls just as if they were regular files. For example, the first argument to the `open(S)` system call can be a full pathname of a special device file (say, `/dev/hfd096ds15`).

There are 2 types of device special files. The "character" or "raw" special file provides access to a *character device driver* (q.v.) and the "block" or "buffered" special file provides access to a *block device driver* (q.v.). A device driver can provide either or both of the "character" and "block" interfaces, and thus there can be both a character special file and a block special file that provide 2 different interfaces to the same device. Typically, disk devices support both block and character interfaces, while, terminal devices provide only a character interface.

Device special files are created with the `mknod(ADM)` command, and are traditionally stored under the `/dev` directory. The on-disk *inode* (q.v.) of a device special file does not point to any filesystem disk blocks, but instead, stores the *major and minor numbers* (q.v.), of the device driver in the first part of the `di_addr` field.

The first character of the `mode` field in the `ls -l` output for a device special file is a "c" for character special files and "b" for block special files.

For example, the following display shows the `ls -l` output for a disk partition:

```
ls -l /dev/fd096ds15
brw-rw-rw-  6 bin      bin      2, 52 Nov 27 10:50 /dev/fd096ds15

ls -l /dev/rfd096ds15
crw-rw-rw-  5 bin      bin      2, 52 Nov 27 09:13 /dev/rfd096ds15
```

In the sample display above, note the "b" and "c" in the left-most column that indicate whether this is a block or character device. Note also the 2, 52 in the listing where the file size would be displayed for a regular file. "2" is the device major number, and "52" is the minor number. Because both listings pertain to the same disk partition, both device special files have the same major and minor numbers. Conventionally, for a device with both interfaces, the filename for the character interface is the same as the filename for the block interface, except that it starts with an "r".

dial.h header file

The `<dial.h>` header file specifies constants, structures and routines used for modem handling.

See also "Header files" in this *Encyclopedia* and the manual pages for `dial(S)` and `undial(S)`.

Directory

UNIX system files are arranged in directories. All UNIX system directories are arranged in a hierarchy that is structured like an upside-down tree. The directory at the top of this hierarchy is called the root directory (as in the “root” of the upside-down tree). The *root* directory is denoted by a slash (/). It contains several files and directories, and those directories contain other files and directories, and so on. All other UNIX system directories and files branch out below the *root* directory.

A directory is a file that contains a list of ordinary filenames and other directory names, and an indication of their locations. Their location is a reference to the *inode table* (q.v.). Each directory entry consists of the filename and inode number. By convention, a directory contains at least two entries, . and .., referred to as “dot” and “dot-dot”, respectively. Dot refers to the directory itself and dot-dot refers to its parent directory except in the case of the *root* directory whose parent is itself.

To access a file by its name, the kernel must search a directory and find its corresponding *inumber* (q.v.). After it obtains the inumber, it finds the corresponding inode, which contains information about the type of the *file* (q.v.), its access permissions, size, time of last modification, the number of links, and the location of the disk blocks that comprise the file data.

Security ramifications of directory permissions

A user must have read permission on a directory to be able to view the names of files and directories located within that directory. A user cannot actually write to a directory, but can add files to and delete files from a directory with system calls (or commands that issue system calls), as long as the user has write and execute permission for the directory. See *Permissions, file access* (q.v.) for more information on directory permissions.

Directory pathnames

Version 4.0 of the SCO UNIX System V/386 Release 3.2 Operating System supports *pathnames* (q.v.) which consist of components, each of which can be a string of 1 to 255 characters (other than the ASCII slash and null). Each component is separated from other components by the slash character. Earlier versions of the operating system allow only up to 14 characters to be used for each component.

Directory size

Avoid directories that are larger than necessary. A directory that contains only as many entries in the directory file as can fit on a single data storage block can be searched very efficiently. If, for example, a data storage block is 1024 bytes in length and each directory entry is 14 characters or less, then a block can hold 64 entries, which allows 2 additional bytes per entry for the *inumber* (q.v.). Directory files are searched sequentially so a directory

containing fewer entries than can fit on a single data storage block will only have to access a single block. Directories that contain only as many entries as fit on the first 10 data storage blocks can still be searched efficiently on average. The search of a directory file becomes inefficient when indirect blocks are used to store directory file entries. This happens when the number of directory entries in a single directory can no longer fit on 10 data storage blocks. It is especially important to keep login directories small, preferably no more than one block.

Note that directories never shrink. Even if you delete files, so that the number of file entries take up less than 10 storage blocks, there is a performance penalty until the directory is reorganized on the hard disk. See the *System Administrator's Guide* for information about reorganizing directories.

Directory operation routines

The following routines provide control over the special files called directories. For a full description of their use, see the `directory(S)` and `getdents(S)` manual pages.

Directory operation routines

Routine	Description
<code>closedir(dirp)</code>	close the named directory stream and free the structure associated with the directory pointer (<i>dirp</i>)
<code>getdents(fildes, buf, nbyte)</code>	read <i>nbyte</i> bytes from the directory associated with <i>fildes</i> and format them as filesystem-independent directory entries in the buffer pointed to by <i>buf</i>
<code>opendir(filename)</code>	open the directory named by <i>filename</i> and associate a directory stream with it; return a pointer to identify the directory stream in subsequent operations
<code>readdir(dirp)</code>	return a pointer to the next directory entry
<code>rewinddir(dirp)</code>	reset the position of the <i>dirp</i> directory stream to the beginning of the directory
<code>seekdir(dirp, loc)</code>	set the position of the next <code>readdir</code> operation on the directory stream named by <i>dirp</i> to the <i>loc</i> location.
<code>telldir(dirp)</code>	return the current location associated with the named directory stream

dirent.h header file

The `<dirent.h>` header file specifies structures and routines used to manipulate *filesystem* (q.v.) independent *directory* (q.v.) entries.

See also “Header files” in this *Encyclopedia* and the manual pages for `opendir(S)`, `readdir(S)`, `telldir(S)`, `seekdir(S)`, `rewinddir(S)` and `closedir(S)`.

dir.h header file

The `<sys/dir.h>` header file specifies include statements for files used to define structures to read a *directory* (q.v.) for backward compatibility with earlier versions of UNIX operating systems.

See also “Header files” and “dirent.h header file” in this *Encyclopedia* and the manual page for `getdents(S)`.

DMA (Direct Memory Access)

DMA permits data to be transferred between memory and a device without the intervention of the system’s central processing unit. DMA is handled by a chip on the main motherboard or on a device controller board.

For the ISA and EISA architectures, each DMA controller chip has eight circuits, called channels, for conveying data. The channels are designated for either reading or writing data, and are designed to send data in 64K units. Channels 0 to 3 are 8-bit channels, and channels 4 to 7 are 16-bit channels. On some earlier models of DMA controllers, only one channel could be used at a time. The DMAEXCL tunable can be used to restrict DMA use to a single channel. This tunable is set using the `configure(ADM)` utility.

DMA channel 4 is reserved for use by the system. Consult the `/usr/adm/messages` file or use the `hwconfig(C)` command to see which channels are in use before assigning a new channel.

DMA is initiated in a device driver by:

1. Allocating space for the received or transmitted data, typically when the system is first initialized.
2. Displaying the selected DMA channel on the system console when the system is booted.
3. Enabling the DMA chip to specify the number of the channel, whether the channel is for reading or writing, the address of the allocated data buffer, the size of the channel (8 or 16 bits), and the name of the device driver interrupt routine that will handle the DMA I/O.

4. For a write request, placing the data in the buffer. Data is sized into 64K units by the device driver before the DMA is performed.
5. Queueing an I/O request for transfer.
6. For a read request, taking the data from the buffer.

When a channel is no longer needed, it is released. The `dmareq` structure contains the information used when enabling a DMA channel. This structure and the constants used to designate channels are defined in the `<sys/dma.h>` header file. Refer to the *Device Driver Writer's Guide* for information on the routines used for DMA.

The DMA channel is specified in the `/etc/conf/cf.d/mdevice` file. Some of the devices using DMA channels are as follows:

Device Driver	Prefix	DMA Channel
Adaptec SCSI controller	ad	5
Cartridge tape	ct	1
Floppy disk	fd	2
Floppy tape	ft	2
Irwin tape	ir	2

Domain

A domain uniquely identifies a machine for receiving or sending electronic mail. A domain includes the name of a machine in an organization, or in a subordinate unit, such as a department, and site's organization or country. The following are types of domains:

Top-level domain

an officially registered name that describes the purpose of a group of institutions or a code that is associated with a country. Examples include *com* for commercial organizations, or *fi* for Finland.

On the Internet, the following common top-level domains are defined:

COM	commercial institutions
<i>code</i>	country code - defined by ISO 3166:1988
EDU	educational and research institutions
GOV	government institutions
MIL	military institutions
NET	network
ORG	organization (generic)
UUCP	an unregistered domain name where users transmit information between cooperating neighbor machines via UUCP

Domain names can be either upper or lowercase; MMDF is case insensitive when evaluating domain names.

Subdomain

an officially registered name that describes a company, department, or any subgroup under a top-level domain, for example, *sco*.

Local domain

a name recognized only within an organization that has meaning only within that organization. For example, a department name such as "engr".

Fully qualified domain

the name of a machine and all other domain names, in the following order:

machine.department.company.top-level-domain

or

machine.company.top-level-domain

An example is:

mama.comp.tortilla.COM

This is read as follows: the *mama* machine at the *comp* machine center at site *Tortilla*, which is a commercial organization (COM).

The following table shows fully qualified domain name usages:

ucsc.ucsc.edu	machine at UC Santa Cruz
seismo.css.gov	a machine at the Center for Seismographic Studies

Refer to the SCO UNIX System V/386 Operating System Administrator's Guide for more information.

DOS and OS/2 cross-development environment

The Development System supports a DOS cross-development environment that enables you to generate DOS binaries from the same source tree as UNIX binaries.

When creating source code for use with both DOS and UNIX binaries, be aware of the following:

- The process ID is still a unique integer in the DOS-OS/2 environment, but does not have the same meaning as in the UNIX System V environment.
- Common routines do not use the parent process, process group, tty group, real user, real group, effective user, and effective group IDs when run under DOS-OS/2. The DOS-OS/2 environment does not support the super user or special processes.
- Filenames in DOS-OS/2 have two parts: a name and a name extension. Names can be any combination of up to eight letters or digits. Name extensions may be any combination of up to three letters or digits, preceded by a period (.).
- Pathnames in DOS can be any combination of directory names separated by a backslash (\). The slash (/) used in the UNIX System V environment is not allowed unless the user has redefined the leading character used with

options in DOS command lines (this character is initially the slash). Directory names may be any combination of up to eight letters or digits. The special names . (dot) and .. (dot-dot) refer to the current directory and the parent directory, just as in the UNIX environment.

- The drive names may be used at the beginning of a pathname to specify a specific disk drive or device. Drive names are generally a letter or combination of letters and digits followed by a colon (:).
- The access permissions in DOS are restricted to read and write by the owner of the file. Because all users own all files in DOS, access permissions do little more than define whether or not the file is a read-only file or can be modified. Execution permission and other permissions defined for files in the UNIX System V environment do not apply to the files in the DOS environment.

For more information, see the (DOS) manual pages.

DSSSL (Document Style Semantics and Specification Language)

The Document Style Semantics and Specification Language (DSSSL) is a language used to describe the formatting of structured documents. This language is the subject of the ISO draft standard 10179.

DSSSL can be used with any document that has a consistent hierarchical structure, but is intended principally for use with SGML (q.v.) documents. It is not a text formatter like *troff* (q.v.), but rather a set of standard, high-level constructs for specifying how a document is to appear when formatted.

Document formatting is described in a DSSSL specification. This specification defines a set of *surfaces*, that is, the logical components of the document in the context of formatting. It gives a mapping between surfaces and the structural elements that already exist in the document (for example, SGML elements). Lastly, attributes describing the page layout can be given for both surfaces and structural elements.

The DSSSL specification and the original document are processed by a DSSSL Language Processor to produce an “augmented” document instance that is passed on to another processor known as a “semantic processor”. The semantic processor is a locally-defined processor that understands the constructs embedded in the augmented document and produces output in the desired format. The output could, for example, be SPDL or another page description language such as PostScript, or even a modified SGML document.

dumprestor.h header file

The `<dumprestor.h>` header file specifies constants, variables and structures that describe the format of the header record and the first record of each description used for dumping files to magnetic tape.

See also “Header files” in this *Encyclopedia* and the manual pages for `xbackup(ADM)`, `xrestore(ADM)` and `xbackup(F)`.

Dynamic library

Dynamic libraries are used by a dynamic linker to satisfy unresolved external references for a program. When a dynamic library is used by a program, all library symbol references are resolved at execution. This will increase performance for the linking stage of a program. However, it may increase the load time for a program, since it is at this stage that all of the library code references are resolved. This contrasts with the methods used by both *archive libraries* (q.v.) and *shared libraries* (q.v.) to resolve external references. The Multics operating systems (a predecessor of UNIX) used dynamic libraries.

NOTE SCO UNIX System V/386 shared libraries are not dynamic libraries. With shared libraries symbols references are resolved at link edit time; but the code is not physically copied.

E

Encryption

Encryption is a method of encoding data so it is harder to decipher. Encryption provides greater security than offered by standard file permissions modes and can be accomplished with two different levels of security and convenience. SCO UNIX System V/386 supports two encryption mechanisms:

- routines in the *libcrypt.a* library, which provide a high level of security but are not as convenient to use.
- routines that access the */bin/crypt* utility, which provide less security but are more convenient to use.

Both of these methods are discussed on the following pages.

*Encryption with **libcrypt.a** (high security with less convenience)*

The *libcrypt.a* routines are listed on the **crypt(S)** manual page.¹

*Encrypting a password with **crypt(S)***

To encrypt a password, use the **crypt(S)** routine. **crypt** takes two arguments: a password and a two-character string called the “salt” value. The salt value is translated to initialize the encryption algorithm in one of 4096 different ways. The encryption algorithm then uses the password to encrypt a constant string. The encryption is repeated until the constant string is thoroughly altered.

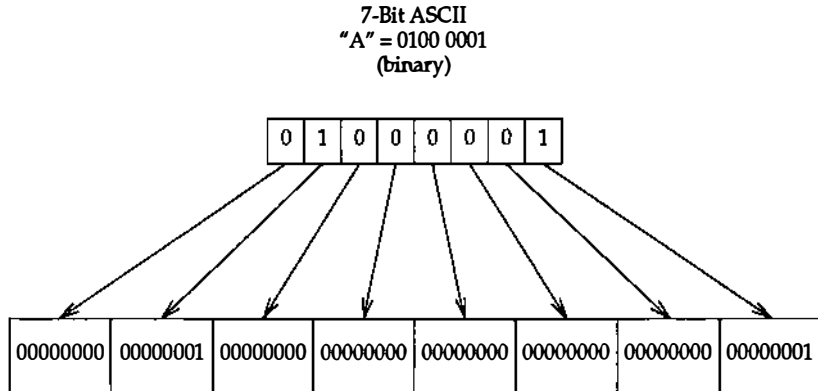
An alternate method is to use the **des_crypt(S)** routine which provides direct access to the Data Encryption Standard (DES) encryption primitive. This routine is used in the same manner as **crypt**.

1. See the “crypt libraries” article for information about *libcrypt.a* software distribution

Encrypting non-password data with *crypt(S)*

To encrypt data that is not specifically a password:

1. take each bit of the data byte and put it into a 8-byte character array as shown in the following diagram:



2. Use the **des_encrypt(S)** routine to pass a pointer to a 64-byte array containing 8 such 8-byte arrays, each of which is an expanded representation of the bits in an 8-bit byte. This pointer is passed along with a flag argument set to a value of zero to specify encryption. The encrypted data is returned in the same block of data that was passed to the routine.
3. Data can be decrypted with **des_encrypt** by supplying a flag value of 1.

The **des_encrypt(S)** routine encrypts 64 bits of data at a time, but the data must be prepared so that each byte value is in the 8-byte array before **des_encrypt** can use it. The **des_setkey(S)** routine sets the 56-bit encryption key to be used by **des_encrypt**.

Preparing data for des_encrypt(S) (Part 1/3)

```

1      #define BITS_PER_BYTE    8          /* bits in a byte */
2      #define BYTES_PER_BLOCK  8          /* size of input block */
3      #define OUTPUT_SIZE      (BITS_PER_BYTE * BYTES_PER_BLOCK)
4      #define IS_BIT_SET(byte,bit) ((byte)&(1<<(bit)))
5      #define SET_BIT(byte,bit)  ((byte) |= (1<<(bit)))
6      #define CLEAR_BIT(byte,bit) ((byte) &= ~(1<<(bit)))
7      expand_bits(unsigned char input[BYTES_PER_BLOCK],
8                  unsigned char output[OUTPUT_SIZE])
9      {
10         int input_bit_index;
11         int input_byte_index;
12         int output_index;
13         unsigned char working_input_bits;
14         output_index = 0;
15         for (input_byte_index = 0;
16             input_byte_index < BYTES_PER_BLOCK;
17             input_byte_index++)
18         {
19             working_input_bits = input[input_byte_index];
20             for (input_bit_index = 0;
21                 input_bit_index < BITS_PER_BYTE;
22                 input_bit_index++)
23             {
24                 if (IS_BIT_SET(working_input_bits, input_bit_index))
25                     output[output_index] = 1;
26                 else
27                     output[output_index] = 0;
28                 output_index++;
29             }
30         }
31     }

```

- line 4 tests bit number 'bit' of byte 'byte' and is 'true' if that bit is set and 'false' otherwise.
- lines 5-6 The following two macros set or clear a bit in a byte. The "byte" argument must be an "lvalue".
- lines 7-31 The output of this is suitable as input to des_encrypt()
- lines 7-8 The **expand_bits** routine accepts an 8-byte buffer as input, then transforms it into a 64-byte array for **des_encrypt**. The resulting 64-byte array contains a representation of the original 8-byte array in which each bit of the input array is represented by a whole byte in the output array. A bit of 0 in the input array is represented by a byte of 0 (0x00) in the output. A bit of 1 in the input array is represented by a byte of 1 (0x01) in the output. The bits of the input are mapped to the bytes of the output in a pure little-endian order. That is, bits in the input are numbered within bytes such that the least significant bit is bit 0, and the most significant bit is bit 7; and bit 0 of byte 1 is considered to be of greater significance than bit 7 of byte 0. See the "Byte" article for more information about little-endian order.

Preparing data for `des_encrypt(S)` (Part 2/3)

```

32     expand_key_bits(unsigned char input[BYTES_PER_BLOCK],
33                     unsigned char output[OUTPUT_SIZE])
34     {
35         int input_bit_index;
36         int input_byte_index;
37         int output_index;

38         unsigned char working_input_bits;

39         output_index = 0;
40         for (input_byte_index = 0;
41             input_byte_index < BYTES_PER_BLOCK;
42             input_byte_index++)
43         {
44             working_input_bits = input[input_byte_index];

45             for (input_bit_index = 0;
46                 input_bit_index < BITS_PER_BYTE - 1;
47                 input_bit_index++)
48             {
49                 if (IS_BIT_SET(working_input_bits, input_bit_index))
50                     output[output_index] = 1;
51                 else
52                     output[output_index] = 0;
53                 output_index++;
54             }
55             output[output_index++] = 0;
56         }
57     }

```

line 32-83 This routine is similar to the one in lines 14-31, except that parity bits (bit 7 of each byte) of the input are explicitly ignored and the corresponding bytes of the output are set to 0. This is useful when setting up an encryption key to pass to the routine `des_setkey(S)`.

lines 32-33 The `expand_key_bits` routine in the example does nearly the same thing as `expand_bits`, except that the parity bit (bit 7) of each input byte is ignored. The result is a 64-byte output array of which 8 bytes are always 0 and 56 bytes have values which correspond to the 56 key bit values. The reason for this format is that it's the way that `des_setkey` wants it.

lines 45-54 Note the limit for the following loop is `BITS_PER_BYTE - 1` rather than `BITS_PER_BYTE` as in the `expand_bits()` routine. This has the purpose of only looking at the 7 data bits of the input bytes.

lines 55 set byte corresponding to parity bit

Preparing data for des_encrypt(S) (Part 3/3)

```

58     contract_bits(unsigned char output[BYTES_PER_BLOCK],
59                 unsigned char input[OUTPUT_SIZE])
60     {
61         int output_bit_index;
62         int output_byte_index;
63         int input_index;

64         unsigned char working_output_bits;

65         input_index = 0;
66         for (output_byte_index = 0;
67             output_byte_index < BYTES_PER_BLOCK;
68                 output_byte_index++)
69             {
70                 for (output_bit_index = 0;
71                     output_bit_index < BITS_PER_BYTE;
72                         output_bit_index++)
73                     {
74                         if (input[input_index++])
75                             SET_BIT(working_output_bits, output_bit_index);
76                         else
77                             CLEAR_BIT(working_output_bits, output_bit_index);
78                     }
79                 output[output_byte_index] = working_output_bits;
80             }
81     }

```

line 58-81 This routine takes output in a format expected from **des_encrypt** and converts it to the form one would normally expect. In other words, it takes a 64-byte block as input, where each byte stores the value intended for a single bit, and packs the data back into an 8-byte block.

Encryption with /bin/crypt (lower security with more convenience)

The **cypher** utility found in */bin/crypt* provides an easier method of encryption, although it is less secure than the DES encryption scheme. Typically, data is read from a file and then encrypted or decrypted using this utility. The advantages of this utility are:

- Encrypting large blocks of data is easier because **write(S)** and **read(S)** are used to pump the data through the encryption mechanism. This avoids having to split the bits in each byte and working with successive 8-byte blocks and so forth as required by the *libcrypt.a* routines.
- The same mechanism is used by **vi(C)** and **ed(C)** to encrypt text files, so encrypted files can be shared between the text editors and your application.

To use the */bin/crypt* utility:

1. Create a 2-integer array and initialize it to a value of zeros. Then pass the name of the array to the **cryptopen(S)** routine. This routine establishes a *pipe* (q.v.) through which the encryption occurs. Two *file descriptors* (q.v.) are returned in the array for reading and writing.
2. Call the **run_setkey(S)** routine to initialize the encryption algorithm. This routine is passed the file descriptors returned from **cryptopen**, and a pointer to a key value.
3. Call the **run_crypt(S)** to encrypt the data. This routine is passed four arguments, an offset into the data buffer, a buffer of data, a count of the number of bytes in the buffer, and the file descriptors from **cryptopen**. The encrypted data can then be read from the pipe using the file descriptors. The **run_crypt** routine can be used to decrypt the buffer given the same arguments.
4. When you are done encrypting, use the **crypt_close(S)** routine to close the pipes.

The following program illustrates use of the **cypher** utility and exercises the **run_setkey(S)**, **run_crypt(S)**, and **crypt_close(S)** library routines. This program is run with an argument that is the name of a file that contains data encrypted with the */bin/crypt(C)* command. Such a file can be created with the **vi -x** command. After working on the file, the output is written to an output file. If a second argument is provided, then the output is written to that file; otherwise it is written back to the original input file, overwriting its original contents.

/bin/crypt utility example (Part 1/2)

```

1      #include <stdio.h>

2      decrypt_input_file(const char *password,
3                          const char *input,
4                          const char *output)
5      {
6          FILE      *infile;
7          FILE      *outfile;
8          int        pipe_fds[2];
9          char       buf[BUFSIZ];
10         int        n;
11         long        offset;

12         infile = fopen(input, "r");
13         outfile = fopen(output, "w");

14         pipe_fds[0] = pipe_fds[1] = 0;
15         run_setkey(pipe_fds, password);

16         offset = 0L;
17         while ((n = fread(buf, 1, BUFSIZ, infile)) > 0)
18         {
19             run_crypt(offset, buf, n, pipe_fds);
20             offset += n;
21             fwrite(buf, 1, n, outfile);
22         }

23         crypt_close(pipe_fds);
24         fclose(infile);
25         fclose(outfile);
26     }

27     encrypt_output_file(const char *password,
28                         const char *input,
29                         const char *output)
30     {
31         decrypt_input_file(password, input, output);
32     }

```

-
- lines 14-15 set up the connection with **/bin/crypt** through a pipe. The file descriptors must be set to zero before starting.
 - lines 16-22 read the input file, process with **run_crypt**, and write to the output file.
 - line 19 **buf** now contains data processed by **/bin/crypt**.
 - lines 23-25 wrap things up with **/bin/crypt** and close the files.
 - lines 27-32 encrypt the file and write it back out. Note that this routine just calls **decrypt_input_file**. A separate routine is used to clearly illustrate the purposes of the routines. The encryption and decryption processes can be identical due to the way the **/bin/crypt** encryption scheme works. This kind of approach doesn't work with other algorithms such as DES or RSA, which use different procedures for encryption and decryption.

/bin/crypt utility example (Part 2/2)

```

33     do_useful_work(const char *file)
34     {
35         char cndbuf[256];

36         sprintf(cndbuf, "/usr/bin/more %s", file);
37         system(cndbuf);
38     }

39     main(int argc, char *argv[])
40     {
41         char      *input_file;
42         char      *temp_file = "/tmp/exampleXXXXXX";
43         char      *output_file;
44         char      *password;

45         if (argc < 2)
46         {
47             printf("usage: %s workfile\n", argv[0]);
48             exit(1);
49         }
50         input_file = argv[1];
51         if (argc >= 3)
52             output_file = argv[2];
53         else
54             output_file = input_file;
55         mktemp(temp_file);
56         password = getpass("Enter file password: ");
57         decrypt_input_file(password, input_file, temp_file);
58         do_useful_work(temp_file);
59         encrypt_output_file(password, temp_file, output_file);
60     }

```

lines 33-38 Do something useful with the decrypted data. As an illustration, we just invoke **more** on the file; a real program would probably do something more specific.

lines 39-60 The **main** routine illustrates the flow of the program

environ variable

In a C *program* (q.v.), the **environ** variable points to an array of pointers to the strings that constitute the *process* (q.v.) *environment* (q.v.). **environ** is initialized from **envp**, the third argument of **main**, by the C runtime startup routine. **environ** is defined in C as

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

The **getenv**(S) and **putenv**(S) routines use the **environ** variable to access and modify the environment array. When **putenv** is called to add or delete environment settings, the environment array changes size. The array's location in memory might also change, depending on the *program's* (q.v.) memory requirements. The **environ** variable is adjusted in these cases and thus always points to the correct array location.

Environment

The user's environment holds information about the current *process* (q.v.). Information about a user's terminal, home *directory* (q.v.), or default shell are a few examples of what is included in a user's environment. The information contained in the user's process environment is used by user applications, by shell scripts, and by shells themselves. The *process* (q.v.) environment is comprised of an array of strings of the form:

name=string

The *name* argument (q.v.) represents the name of an environment *variable* (q.v.) that can be either a user-specific variable or one of the system-defined variables. The *string* argument represents the value of the corresponding variable.

Commonly defined environment variables

Some of the more common system variables with their meanings follow:

EDITOR	Defines the user's default editor
HOME	Defines the user's login directory
MAIL	Defines the file where a user's mail is delivered <i>utility</i> (q.v.)
PATH	Defines the search path for directories containing executables
SHELL	Defines the user's default shell
TERM	Defines the type of terminal being used
TZ	Defines the user's local <i>time</i> (q.v.) zone information

Generally, the system-defined variables are initially set in */etc/profile*, */etc/login*, or */etc/cshrc*. When a new user is added, the default values set in one of these *files* (q.v.) comprise the new user's environment. Environments are then inherited from parent processes.

Manipulating environment variables

You can change the value of a system-defined variable or include newly defined variables with different names. Locally redefined system variables override default system definitions.

To alter or add environment variables at the shell level, setting the variables in either the *.profile* or the *.login* in the home directory automatically defines these variables upon login. If you are using the *Bourne shell* (q.v.) or *Korn shell* (q.v.), variables must be exported to make them available to the environments of commands. The *environ(M)* manual page describes how to change or define a new environment variable. Locally defined system variables set in this manner override default system definitions.

To alter or add environment variables at the program level, use the *setenv(S)* and *getenv(S)* routines.

eqn

eqn is a *preprocessor* (q.v.) used with the text-formatter *troff* (q.v.) to typeset mathematics. *eqn* supports a notation for describing mathematical expressions. The user embeds instructions in a *troff* document. The *file* (q.v.) is processed by *eqn*, which converts these *commands* (q.v.) into the equivalent *troff* codes. The resulting file is then processed by *troff*.

Ordinary alphabetic characters and common mathematical operators can be included literally in equations. *eqn* supports a number of special features. Greek characters are available. In addition, users may define their own symbols in terms of *troff* primitives (some vendors provide sets of these definitions). *eqn* recognizes keywords such as *cos* and *lim* and prints them in an appropriate font. It supports such mathematical constructions as subscripts and superscripts, brackets, square roots, fractions, matrices, diacritical marks, summation, and integration.

troff and *eqn* are not included in the SCO UNIX System V/386 Development System, but are available from a number of outside vendors or as public domain software for UNIX, XENIX, and DOS systems. For more information about *troff*, refer to O'Reilly and Associates *UNIX Text Processing*.

errno.h header file

The `<errno.h>` header file specifies variables and routines used to report errors during the execution of a program.

See also “Header files” and “Error handling” in this *Encyclopedia*, the manual pages for `intro(S)` and `perror(S)` and Maurice Bach’s *Design of the UNIX Operating System*.

Error handling

Error handling refers to coding that detects and reacts to system error conditions and application-specific errors. Proper error handling increases the robustness and usability of your application. In general, your code should include error handling sections for any condition that could occur. Sometimes when performance is more important than robustness, you may choose to omit error handling code for conditions that are possible but unlikely; each error check takes a machine instruction or two, so for error conditions that rarely occur and are not dangerous when they do, omitting the error handling can enhance performance. In general, however, we do not recommend omitting error handling code, especially if you are not sure you understand all ramifications of the unmanaged error condition, and if you are not the only one who will use the code you are creating. Error handling is as much a part of documenting code as it is a means of tracking errors.

System error handling

Errors generated by system calls originate in the operating system or from a *device driver* (q.v.). In either case, an error value defined in *errno.h header file* (q.v.) is assigned to the `u_error` member of the *kernel* (q.v.) *user structure* (q.v.). Then the operating system automatically conveys this error value to the user-level `errno` constant.

If a user-space program has `errno` declared, then the value can be tested and acted on. Even without `errno` set, if an error occurs in a system call, the return value contains a negative number that can be tested.

The following code fragment illustrates how **errno** and return value testing is used:

Error handling example

```

1      #include <errno.h>
2      #include <fcntl.h>
3      #include <unistd.h>
4      extern int errno;
5      int fd;
6      char *filename;
7      main(argc, argv)
8      int argc;
9      char *argv[];
10     {
11         extern void exit(), perror(), printf();
12         if (argc < 2) {
13             (void) fprintf(stderr, "usage: %s filename\n", argv[0]);
14             exit(1);
15             filename = argv[1];
16             if ((fd = open(filename, O_RDWR)) < 0) {
17                 perror("open failed");
18                 exit(2);
19             }
20             (void) lseek(fd, 0, 0L);
21             if (lockf(fd, F_TEST, 0L) < 0) {
22                 switch (errno) {
23                     case EACCES; /* A lock is set */
24                     case EAGAIN;
25                         (void) printf("file is locked by another process\n");
26                         break;
27                     case EBADE; /* bad argument passed to lockf */
28                         perror("lockf bad argument");
29                         break;
30                     default;
31                         (void) printf("lockf: unknown error <fd>\n", errno);
32                         break;
33                 }
34             }
35         }

```

- | | |
|----------------|--|
| lines 1 to 3 | Include <i>header files</i> (q.v.). <i>errno.h</i> provides the error values, such as EACCES . <i>fcntl.h</i> provides the open flag O_RDWR , and <i>unistd.h</i> is used for <i>file and record locking</i> (q.v.). |
| line 4 | Declare errno for use in the program. This lets the program test the error values stored by the system in this <i>variable</i> (q.v.). |
| lines 12 to 14 | If the program is executed without <i>arguments</i> (q.v.), then the usage message is displayed on stderr using fprintf(S) . Then exit(S) is called with the argument 1 to indicate to the caller that an error occurred. Use of exit to pass numeric error values requires informing the user about the significance of the error in your program documentation. |

- lines 16 to 18 A file is opened. If the **open(S)** system call returns an error as a value of -1, then **perror(S)** is called to display an error string.
- lines 20 to 34 The purpose of this code is to determine if a file is locked during a file and record locking procedure. This type of record locking is used for *XENIX cross development* (q.v.). Line 20 seeks to the beginning of a file (q.v.). Line 21 sets the size of the test region to test the file for a lock from the beginning to the end of the file. An error value indicates the presence of a lock. Line 22 tests the value for **errno**. If the value is **EACCES**, line 23, or **EAGAIN**, line 24, then the file is locked. In line 25, **printf(S)** is used to display an error message. Lines 27 to 29 test for a bad argument to **lockf** displaying a message if an error occurs. Lines 30 to 32 check for an error value other than **EACCES**, **EAGAIN**, or **EBADF** and displays a message if such an error is received.

Application error handling

Application error handling is the handling of errors that are related to specific operations of the application, the state of application data, and so forth. For instance, in an application that looks up names and addresses, finding that there is no one with the specified name in the database is an application error.

Application errors are handled similarly to system errors: the code should print a message to the user and exit, abort, return, or initiate some application-specific recovery. depending on the circumstances of the error. The **printf(S)** and **fprintf(S)** functions can be used to print a literal message to the user. We recommend that such messages be implemented using *message catalogs* (q.v.); with message catalogs, the error messages can be translated to a different language without recompiling the application code.

Shell script error handling

In a *shell* (q.v.) script, the **echo command** (q.v.) is used to display messages and the **exit command** can be given a numerical argument in the range of 0 to 255 so that specific error events can be identified. All shell scripts should have an exit status, though the shell does not enforce this. The exit status of the most recently terminated script is available in the **\$?** *environment* (q.v.) variable. The **if command** can test the exit status of a command, or a list of commands (which may contain pipelines). For example:

```
if grep "#define" *.h | awk '{print $4}'; pwd
then
.
.
.
fi
```

The following example shows how error handling occurs in a shell script. The **test** command is a special command that evaluates a conditional expression.

Shell script error handling

```

1      if test "$#" -lt 1 ; then
2          CMD='basename $0'
3          echo "rename script usage: $CMD oldfile newfile"
4          exit 1
5      fi
6      if test ! -f "$1" ; then
7          echo "input file <$1> not found"
8          exit 2
9      fi
10     if test -f "$2" ; then
11         echo "new filename <$2> already exists"
12         exit 3
13     fi
14     mv $1 $2
15     exit 0

```

-
- | | |
|---------------|--|
| line 1 | This shell script renames a file after ensuring that the input file exists and the output file does not exist. |
| lines 2 to 6 | A usage message is displayed if arguments are not entered when the shell script is executed. Line 3 gets the name of the shell script. The basename(C) utility is used to cover the possibility that the shell script was executed by specifying the file <i>pathname</i> (q.v.). Line 4 displays the usage message, and in line 5 the shell script exits. The exit value 1 indicates an error occurred. If this shell script was called from another script, the shell variable \$? would indicate the value returned from the exit statement. |
| lines 7 to 10 | If the file to be renamed does not exist, a message displays and another exit value is generated. |
| lines 8 to 11 | If a file exists with the same <i>filename</i> (q.v.) as the specified new filename, an error message displays and another exit value is generated. |
| line 12 | The input file is renamed to that of the specified new filename. |
| line 13 | The shell script exits with a successful completion exit code. exit 0 is the same as exit without a value specified, and has the same effect as not specifying an exit command at all. When the exit command is not specified at the end of a script, the script ends and returns 0. |

eti.h header file

The *<eti.h>* header file specifies constants and routines used by the *Extended Terminal Interface* (ETI) (q.v.).

See also “Header files” and “Curses” in this *Encyclopedia* and the *User Interfaces Guide* manual.

Effective UID (EUID) and Effective GID (EGID)

The effective user ID (EUID) and effective group ID (EGID) of an active process determine the owner and group assigned to newly-created files. When accessing a file, if the file owner matches the EUID, the “user” permission bits determine the permitted access. If the EUID does not match the file owner but the EGID matches the file group owner, then the “group” permission bits determine access. Otherwise, the “other” bits determine access.

The EUID and EGID also determine a process’s ability to access the IPC resources (shared memory regions, message queues, and semaphores). Finally, the EUID sending a signal must match that of the target process (or be superuser) for the system to allow the signal to be sent.

The EUID and EGID are inherited from the parent process when a process forks. The EUID and EGID are the same as the real user and group ID unless the process is descended from a process that called *exec(S)* on an object file that had the *setuid* or *setgid* bit set in its permission modes. See also the *exec(S)*, *getegid(S)*, *geteuid(S)*, and *setuid(S)* manual pages, the “Permissions, file access”, and “Signals”, articles in this *Encyclopedia*, and Maurice Bach, *Design of the UNIX Operating System* for more information.

Executable file

An executable *file* (q.v.) is a file, containing instructions, that can be executed by the computer. That is, when the *filename* (q.v.) is typed at a UNIX System prompt, the commands in the file are executed. Executable files can be object files or files of shell *commands* (q.v.). For a file to be executable, it must have execute *permission* (q.v.). You can give a file execute permission with the *chmod(C)* command. Files with execute permission that are not valid executable object files or shell command files will not execute; text files that have been given execute permission but which are not valid shell command files will produce unexpected results.

exit

exit, C function

The C function call `exit(n)`, where *n* is an integer-valued expression called the “exit status,” can be used at any point within a program to explicitly terminate the program. The `return` statement, if it appears in the `main` routine, also terminates the program. `exit` also performs some clean-up operations such as closing *file descriptors* (q.v.). When a C program is compiled and linked with `cc(CP)` the resulting object code contains a call to `exit`. The form of this call is:

```
exit(main(argc,argv,envp));
```

The exit status will be the value returned by `main`. If the program is linked with `ld(CP)` directly, then the C runtime startup routine (*/lib/crt0.o*), which includes the definition of `exit`, is not linked automatically. In this case, the programmer must call `exit` explicitly somewhere in the program. Linking without the runtime startup routine is not recommended.

exit, shell command

The shell command `exit` is used to exit from a shell program. If it is executed inside a shell function, the shell program that called that function is also terminated. If `exit` is used on the shell command line it will cause that shell to be terminated, which may result in the user being logged off. The general format of this command is:

```
exit n
```

The integer-valued expression *n* is optional; it is the “exit status” of the program. By convention, an exit status of zero indicates successful termination.

Extended terminal interface (ETI)

ETI, the Extended Terminal Interface, is a set of C *library routines* (q.v.) that promote the development of application *programs* (q.v.) that display and manipulate *windows* (q.v.), panels, menus, and forms and run under the UNIX System.

ETI consists of five libraries: (`curses`), `panel`, `menu`, `form`, and the TAM transition library. These routines are C *functions* (q.v.) and *macros* (q.v.); many of them resemble routines in the standard C library.

One major feature of ETI is cursor *optimization* (q.v.), which minimizes the amount a cursor has to move around a screen to update it. The other major feature is terminal independence. When a program that uses ETI begins execution it searches one of two terminal description databases, called *-terminfo* (q.v.) and *termcap* (q.v.), in order to find the correct description for the terminal. The ETI program uses this description when interacting with the terminal. This allows a program to run on many different terminals without being concerned about the specific features of each. See *User Interfaces Guide* for specific information on ETI.

F

fatal.h header file

The *<fatal.h>* header file specifies constants, macros and variables for use with the Programmer's Workbench (libPW).

See also "Header files" in this *Encyclopedia*.

fcntl.h header file

The *<fcntl.h>* header file specifies files to include to specify the constants and structures used by the *file access system calls* (q.v.) and the *file and record locking* (q.v.) routines.

See also "Header files" in this *Encyclopedia* and the manual pages for **open(S)** and **fcntl(S)**.

Field

A field is a logical structure that contains data that defines one attribute of an entity. A collection of fields is called a *record* (q.v.).

UNIX system files do not directly support the concept of fields and records, but an application program can impose such a structure itself. For instance, *ISAM files* (q.v.) use fields to denote a data category, such as a name or an address. Also, the **sort(C)** and **cut(C)** commands operate on fields and allow the field *delimiter* (q.v.) to be specified.

FIFOs (named pipes)

See "Pipes."

File

A file is the basic data unit stored on a disk or accessed by a program. The system supports several types of files:

regular or ordinary	contains any kind of data, including ASCII data, shell scripts, ISO 8889 data, or binary executable code
directory	contains the names and inode numbers of all files located in that directory
special device	usually located in the <i>/dev</i> directory, represents devices and pseudo-devices. Instead of the size member of the inode, special device files have a major and minor number that are used to access the appropriate driver.
FIFO (pipe) file	created with the <code>mknod(ADM)</code> command or <code>mknod(S)</code> system call and used to transfer data between two concurrently executing programs

The operating system does not impose a structure on files other than directories. Control information for the file is stored in the *inode* (q.v.) for the file; the actual name of the file is stored in a corresponding entry in the appropriate *directory* (q.v.) file. The system uses the file's *inumber* (q.v.) to associate the file's inode with the filename in the directory.

Basic file access

Files can be accessed with either the *standard I/O* (q.v.) library routines or the *file access system calls* (q.v.).

- File access system calls access files through a *file descriptor* (q.v.). The `open(S)` and `creat(S)` system calls return a file descriptor that is used as an argument to the other system calls.
- Standard I/O library routines interact with a data stream identified by a file pointer of type `FILE` defined in the *<stdio.h> header file* (q.v.). The data stream structure identifies the file descriptor as well as the location of the *buffer* (q.v.), the current character position in the buffer, and whether the file is being read or written. The `fopen(S)` routine returns the `FILE` pointer that indicates the data stream; all other routines use the data stream as an argument.

The “Standard I/O” article gives more information about the system calls and library routines that are used to access files. See Marc Rochkind's *Advanced UNIX Programming* and Kernighan and Ritchie's *C Programming Language* for detailed information and examples of using the file access system calls and library routines.

Maximum number of open files

The number of files a process may have open is a configurable number. The current value of this parameter can be obtained on POSIX-compliant systems by using the `sysconf(S)` system call. The `OPEN_MAX` symbol in the *<limits.h>*

header file gives the maximum number of open file descriptors for POSIX-compliant systems.

Historically, UNIX operating systems have provided for at least 20 open files, and have defined `NOFILE` as the actual number; POSIX requires that at least 16 file descriptors can be opened for a process at one time, and ANSI C requires at least 8. On SCO UNIX System V/386, the number of open files allowed may range between `NOFILES_MIN` and `NOFILES_MAX` (defined in the `<sys/param.h>` header file), and is adjusted by the `configure(ADM)` or `idtune(ADM)` commands. The obsolete symbol `NOFILE` is also defined, but should be used only on XENIX and other non-POSIX systems.

The following code fragment defines `max_open()` to return the current value for the maximum number of open files.

```

1 #include <unistd.h>

2 #ifdef _POSIX_VERSION
3 # include <limits.h>
4 # ifdef OPEN_MAX
5 #     define max_open() OPEN_MAX
6 # else
7 int
8 max_open(void)
9 {
10     return sysconf(_SC_OPEN_MAX);
11 }
12 # endif
13 # else
14 # include <sys/param.h>
15 # ifdef NOFILE
16 #     define max_open() NOFILE
17 # else
18 #     error Cannot determine NOFILE for this system
19 # endif
20 #endif

```

lines 1-13 determine the maximum number of open files on a POSIX-compliant system

lines 14-21 determine the maximum number of open files on non-POSIX-compliant systems. This code might require modification to run on some non-POSIX-compliant systems.

File access system calls

File access system calls are entry points to the kernel that allow access to files and peripheral devices. These system calls give a program access to files and devices in ways that *standard I/O* (q.v.) routines do not. The file access system calls do not provide buffering or other useful services provided by the standard I/O package, so programs that use these calls must provide these services.

The file access system calls use the *file descriptor* (q.v.) pointer to identify the file.

File access system calls

Routine	Description
<code>open(path,oflag [,mode])</code>	opens a file descriptor for the file pointed to by <i>path</i> and sets the file status flags according to <i>oflag</i>
<code>creat(path,mode)</code>	creates a file or prepares to rewrite an existing file pointed to by <i>path</i> ; if the file does not exist, <i>mode</i> determines the low-order 12 bits of file mode
<code>close(fildes)</code>	closes the file descriptor indicated by <i>fildes</i>
<code>read(fildes,buf,nbyte)</code>	reads <i>nbyte</i> bytes from the file associated with <i>fildes</i> into the buffer pointed to by <i>buf</i>
<code>write(fildes,buf,nbyte)</code>	writes <i>nbyte</i> bytes from the buffer pointed to by <i>buf</i> to the file associated with <i>fildes</i>
<code>lseek(fildes,offset,whence)</code>	repositions a file pointer associated with <i>fildes</i> ; if <i>whence</i> is 0, the pointer is set to <i>offset</i> bytes; if <i>whence</i> is 1, the pointer is set to its current location plus <i>offset</i> ; if <i>whence</i> is 2, the pointer is set to the size of the file plus <i>offset</i>
<code>dup(fildes)</code>	creates a second descriptor for a file with the same access mode, file pointer, and open file; <i>fildes</i> is obtained from the <code>open</code> or <code>creat</code> system call
<code>dup2(fildes,fildes2)</code>	reassigns the file descriptor <i>fildes2</i> to the same file as <i>fildes</i>

SCO UNIX System V/386 provides both library routines, as part of the Standard C Library, and system calls for accessing files. For more information about file access library routines provided in the Standard C Library, refer to the "Standard I/O" article in this *Encyclopedia*.

File and record locking

File and record locking prevents one process from accessing a record while another process is updating it. Three distinct mechanisms are supported for implementing file and record locking:

- UNIX system file and record locking (COFF executable files only)
- XENIX file and record locking (OMF executable files only)
- ISAM file and record locking

UNIX system and XENIX file and record locking

There are two types of UNIX system and XENIX file locking: mandatory and advisory. Under mandatory locking, the system prevents processes from reading or writing to a file that is locked by another process; the standard I/O subfunctions and I/O system calls enforce the locking protocol. Advisory locking can coordinate self-synchronizing processes, but under advisory locking, the system does not prevent reading or writing and users are expected to cooperate. Advisory is the default form of file locking.

XENIX file and record locking

To determine which locking technique to use when performing *XENIX cross development* (q.v.), follow these guidelines:

- Mandatory locking occurs on all XENIX systems prior to 2.3. For release 2.3, mandatory locking occurs if you:
 - a. use the `locking(S)` system call, and
 - b. do *not* set the advisory locking flag in the *x.out* header, or
 - c. set the advisory locking flag in the *x.out* header, but the access mode of the file requires mandatory locking to be enforced (see the `chmod(C)` manual page).
- Advisory locking occurs on UNIX systems and XENIX 2.3 if you:
 - a. use the `lockf(S)` or `fcntl(S)` system calls, and
 - b. run a binary with the advisory locking flag in the *x.out* header, and
 - c. the access mode of the file does not require mandatory locking to be enforced (see `chmod(C)`).

Routines for XENIX file locking

Routine	Description
<code>fcntl(fildes,cmd,arg)</code>	controls open files, allowing reads or writes of locked areas; <i>fildes</i> is obtained from a creat , open , dup , fcntl , or pipe system call; <i>arg</i> is an int or pointer, depending on the <i>cmd</i> .
<code>lockf(fildes,function,size)</code>	locks the file specified by the <i>fildes</i> ; <i>function</i> specifies what action to take; <i>size</i> specifies the number of contiguous bytes to lock or unlock.
<code>locking(fildes,mode,size)</code>	locks or unlocks regions of a file specified by <i>fildes</i> for reading and writing; <i>mode</i> is an integer value that defines the type of lock applied to the file; <i>size</i> is a long integer value that specifies the number of bytes to lock or unlock. locking is only available to programs compiled with the -lx option.

ISAM file and record locking

The most powerful system of file and record locking supported works with *ISAM files* (q.v.). Three types of file locking are supported:

- exclusive** Other processes cannot read or write the file or obtain locks on it until it is closed.
- manual** Other processes can read the file, but they cannot write to the file or obtain locks on it. When you specify a manual lock at the file level, you can explicitly lock the file later with an **islock** call and unlock it with an **isunlock** call.
- shared** Other processes can read the file and acquire shared locks on it, but they cannot change the data or lock the file exclusively until it is closed.

File descriptor

When a process first accesses a file with the **open(S)** or **creat(S)** system call, it is assigned a small integer value, the file descriptor for that file. All subsequent file I/O uses this descriptor, which serves as a sort of handle on the file for the process.

Each executing process has a file descriptor table that contains an entry for each open file, the table is indexed by the file descriptor. These entries contain pointers to the system file table, which in turn contain pointers to the system inode table. File descriptors have values from 0 to (n-1), where *n* is the maximum number of open files allowed per process. See the "File" article for information about determining this value.

I/O system calls other than **open** and **creat** use the file descriptor to specify the target file. Because the descriptor table is part of a process's environment, a *child process* (q.v.) inherits a copy of its parent's file descriptor table. The **close** system call notifies the system that no more I/O will be done with the referenced file descriptor and frees its slot in the descriptor table for reuse.

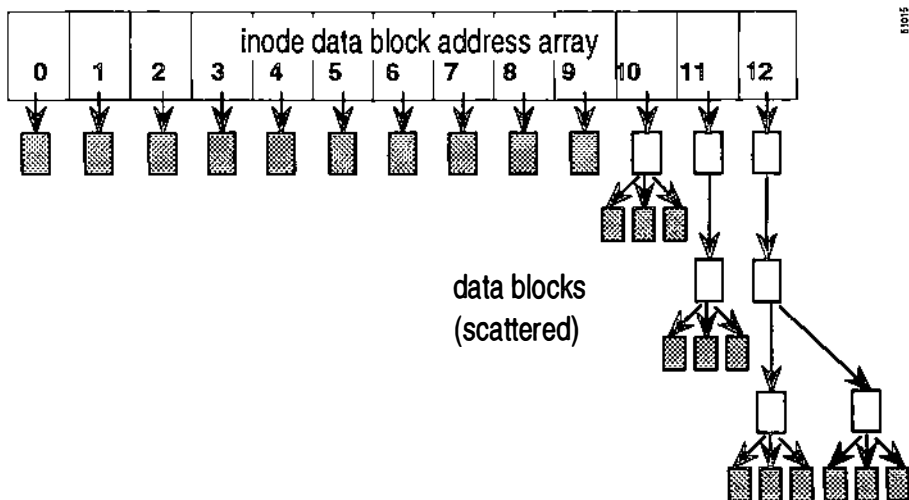
Conventionally, descriptor 0 refers to standard input (*stdin*), descriptor 1 refers to standard output (*stdout*), and descriptor 2 refers to standard error (*stderr*). For regular files, each **open/creat** operation creates one new file descriptor; the **pipe(S)** system call creates two descriptors (one for the read end and one for the write end of the pipe). The **dup(S)** and **dup2(S)** system calls duplicate a file descriptor. In other words, they create a new file descriptor that represents the same file or pipe, same file pointer, and same access mode as the original file descriptor.

File structure

Each file in the *filesystem* (q.v.) is associated with an *inode* (q.v.). The inode for a file contains a table that points to the data storage blocks associated with the file. This table contains 13 entries:

- entries 0-9 point to the first 10 data storage blocks associated with the file
- entry 10 for files that are larger than 10 blocks, points to an indirect block, which contains addresses of the next 256 data blocks associated with the file
- entry 11 for files that contain more blocks than can be accessed through the single indirection of block 10, a double indirect block that contains the addresses of 256 indirect blocks, each of which contains the addresses of another 256 data blocks
- entry 12 for even larger files, points to a triple indirect block that contains the addresses of a set of double indirect blocks that point to sets of indirect blocks that point to the data blocks

The structure of a file is illustrated in the following figure:



Accessing data blocks

Files are seldom large enough to require double and triple indirection, but if these are used, one can incur a fair amount of overhead for file access operations.

File table, system

The system file table is a *filesystem access table* (q.v.) that contains information about every open file on the system. Each time a file is opened, an entry is allocated and populated in the system file table. A system file table entry contains the following information:

- reference count number of *file descriptors* (q.v.) that point to this entry. These may be from one or more processes. If two processes are pointing to the same system file table entry, they must be related either through a *fork(S)* call, where the file must have been opened before the *fork*, or through a *dup(S)* call.
- current offset user's position within the file, in bytes
- flags how the file was opened. They include the following values:
 - read file can be accessed for reading
 - write file is available for writing
 - append before each write to the file, the current offset is set equal to the size of the file. Therefore, if two or more users are appending to the same file, they always append to the end of the file.

no delay If the file being opened is a named pipe, reads to an empty pipe do not cause the process to be put to sleep, nor do writes to a full pipe. Instead, an error is returned.

pointer to system inode table entry

link to the remainder of the information about the file; an access pathway to the data blocks for the file

The **file** command of the **crash**(ADM) command displays the system file table. Use the inode number obtained from the **inode** command of **crash** to locate the file's entry in the file table. Using the assignment of slots in the system file table and the system inode table entry associated with each of the user's file descriptors, you can determine which files and devices are in use (or were in use when the system crashed).

Each slot displays the number of file descriptors open against the file (in other words, the number of **open**(S), **creat**(S) or **dup**(S) calls that have been issued against the file without a corresponding **close**(S) call); the slot of a corresponding inode in memory; the current file offset (or **lseek**(S) value); and the access modes that were specified when the file was opened.

filehdr.h header file

The *<filehdr.h>* header file specifies constants, macros and structures used for COFF (Common Object File Format) (q.v.) headers.

See also "Header files" in this *Encyclopedia*.

Filename

Strings of characters are used to name an ordinary file, a special file, or a directory. Version 4.0 of the SCO UNIX System V/386 Release 3.2 Operating System supports file names that are strings of 1 to 255 characters. These characters may be selected from the set of all character values excluding 0 (null) and the ASCII code for a slash (" / "). Earlier release of the system support filenames up to 14 characters.

It is generally unwise to use "*", "?", "[", and "]" in filenames because of the special meaning attached to these characters by the shell. The high order bit of the character also should not be set.

Filesystem access tables

Three system tables maintain information about all files that are opened or referenced. These are:

Inode Table

contains information from the disk inodes for each open or referenced file. The operating system maintains one inode table that is shared by all processes; it is part of the operating system's address space.

File Table

contains information about open files. Each time a file is opened, an entry is allocated in this table to identify the way the file was opened and the user's current offset in the file. This table is part of the operating system's address space.

File Descriptor Table

a table of pointers to entries in the system file table. One table for each process resides in the u-area in the system address space.

See the "Inode table," "File table," and "File descriptor" articles in this *Encyclopedia* for more information about the contents of these tables.

To illustrate how the file access tables work, the following sections describe the internal system activities related to the **open**, **creat**, **read**, and **write** system calls.

Internal system activities for open

The **open** system call opens an existing file and returns a file descriptor. As an example, assume that the argument to **open** is the path */etc/motd*:

1. The operating system sees that the pathname starts with a slash indicating the *root* inode.
2. Using the *root* inode, the system does a linear scan of the *root* directory file looking for an entry *etc*. When *etc* is found, the operating system picks up the *inumber* (q.v.) associated with *etc*.
3. The *inumber* gives the offset into the *ilist* (q.v.) in which the inode for *etc* is stored. At that location, the system determines that *etc* is a directory by looking at the file type.
4. The *etc* directory is searched linearly until the *motd* entry is found.
5. The *motd* is found, and its *inumber* is picked up and used as an index into the *ilist* (q.v.) to find the inode for *motd*.
6. The inode for *motd* is copied to the system inode table, if it is not already in the inode table, and the reference count is incremented.
7. The system file table entry is allocated; the pointer into the system inode table is set; the offset for the I/O pointer is set to zero to indicate the file's beginning, and the reference count is initialized to one.

8. The user area file descriptor table entry is allocated with a pointer set to the entry in the system file table.

The algorithm for locating the a file's inode illustrates why it is advisable to keep directories small. Search time also is speeded up by keeping subdirectory names near the beginning of a directory file, which can be accomplished by the `dcopy(ADM)` command.

Internal system activities for creat

The `creat(S)` system call creates a new file and returns a file descriptor. It functions like the `open` system call, with three additional steps at the beginning:

1. A free on-disk inode is obtained.
2. The file's mode is determined and entered in the inode.
3. The system goes through a directory search similar to that used in the `open` system call, except that here the last element of the pathname and the corresponding *inumber* (q.v.) are written by the system into the directory that is the next-to-last portion of the pathname.

Internal system activities for read and write

The `read(S)` and `write(S)` system calls take a *file descriptor* (q.v.) as an argument, and follow these steps:

1. Using the file descriptor as an index, the file descriptor table is accessed to get a pointer to the system file table.
2. The user buffer address and number of bytes to read/write are supplied as arguments to the call. The correct offset into the buffer is obtained from the system file table entry.
3. The inode is found by following the pointer from the system file table entry to the system inode table.
4. The system checks whether any locks are held on the file.
5. For read operations, the operating system copies the data from storage to the user's buffer and performs the read provided the file is not locked by another process.
6. For write operations, the operating system copies the data from the user's buffer into the data blocks. If new blocks are needed, the system obtains them from the filesystem's list of free data blocks. The write operation takes place provided the file is not locked by another process.
7. Before the system call returns to the user, the number of bytes read or written is added to the offset in the system file table.

Files shared by related processes

If related processes are sharing an open file (as is the case after a fork), they also share the entry in the system file table. Consequently, they share the file pointer.

Unrelated processes that access the same file have separate entries in the system file table. Because they have separate file offsets, they may read from or write to different places in the file.

The correct offset at which the read or write operations should take place is tracked by the offset entry in the system file table. In both cases, the entry in the system inode table is shared.

Pathname conversion

The directory search and pathname conversion take place only when the file is opened. For subsequent access of the file, the system supplies a file descriptor which is an index into the file descriptor table in your user process area. The file descriptor table points to the system file table entry where the pointer to the system inode table is picked up. Given the inode, the system can find the data blocks that make up the file.

Filesystems

A filesystem organizes a collection of files and directories that are stored on one partition of a disk. The Release 3.2 operating system supports four filesystem architectures: S51K, XENIX, ACER Fast Filesystem (AFS), and DOS. The architectures of the first three are described here; the DOS filesystem architecture mimics the filesystem architecture on DOS-OS/2 machines.

The internal structure of a filesystem contains these major sections:

- | | |
|---------------------|---|
| super block | contains global information about the filesystem, such as the filesystem architecture being used, the size of the inode array, how many inodes are being used, the logical block size, the total number of blocks in the filesystem, and the number of blocks being used. It also contains some status information. See "Super block" for more information. |
| ilist | array of inodes for all files in the filesystem. An inode that holds control and status information is associated with each file in the filesystem. The "inumber" is the index into this array. |
| data storage blocks | storage for the data in the files and directories that are part of the filesystem. These are referenced by number and maintained through the inodes of the files that use them. See "File structure" for more information. |

The filesystem architectures are specified in files in the `<fs>` directory:

Filesystem specification files in `/usr/include/sys/fs` directory

File	Filesystem architecture
<code>f5fileys.h</code>	S51K and AFS
<code>xxfileys.h</code>	XENIX
<code>dosfileys.h</code>	DOS

When a filesystem is identified to the operating system with a `mount(ADM)` command, the operating system makes an entry in the mount table and reads the *super block* (q.v.) into an internal buffer maintained by the kernel. Parts of the superblock needed in memory are the lists of free *inodes* (q.v.), and the flags and time fields which are constantly being modified.

Filter

A filter is a *program* (q.v.) that reads information from standard input, acts on it in some way, and sends its results to standard output. It is called a filter because it can be used to transform data in a “pipeline”. Filters are different from editors, which also transform data, and other *commands* (q.v.) because filters do not modify the contents of any *file* (q.v.). Filters can be existing UNIX commands such as `grep`, `cut`, `paste`, `sort`, and `uniq` or can be user written programs that use the same principles of input and output. The following is one example of how filters can be used:

```
$ cat /etc/passwd | grep /bin/ksh | cut -d: -f1 | sort
```

This example starts by listing the contents of the password file and through successive filtering outputs a sorted list of those users that use the *Korn shell*, *ksh* (q.v.), as their login shell. The filters in this example are `grep`, `cut` and `sort`. The command `cat /etc/passwd` lists the contents of the password file which is piped to the command `grep`. `grep` selects those lines containing the string `/bin/ksh`, which refers to the Korn shell. These lines are piped to the filter `cut` which removes all but the user name from each of the remaining lines and *pipes* (q.v.) it to `sort`. `sort` simply sorts the remaining list of names and writes it to standard output.

`sed` and `awk` are also filters but are called programmable filters or data transformers because a list of transformations, in the case of `sed`, or a program, in the case of `awk`, can be supplied as an additional *argument* (q.v.). These allow a greater degree of sophistication and diversity in the transformations that the filters perform.

float.h header file

The `<float.h>` header file specifies constants that define limits for double and float data types.

See also “Header files” in this *Encyclopedia*.

Flow control

Flow control is used to control the data flow on a serial line, output displayed on a terminal screen or in a terminal window, for example. A user can manage flow control by using the `<Ctrl>q` and `<Ctrl>s` keys while data is being received at the terminal. Pressing `<Ctrl>s` sends an ASCII control code to the computer, which stops a scrolling display from continuing; pressing `<Ctrl>q` restarts the scrolling. The `<Ctrl>q` and `<Ctrl>s` characters are *ASCII (q.v.) control characters (q.v.)* known as device controls (DC1 and DC3) or as transmission controls (XON and XOFF). DC3 (19).

From a system administrator's perspective, the ability of a user to be able to exercise flow control is determined by the settings of the `stty(C)` utility. While the default settings permit standard flow control characters like `<Ctrl>q` and `<Ctrl>s`, the ability to use these characters can be restricted.

From a programmer's perspective, a program can call `tcflow(S)` to cause a terminal device display to start or stop accordingly. To start the display, the POSIX constant `TCOON` is used with `tcflow`, to stop the display, `TCOOFF` is used. To send the STOP character to the device, `TCIOFF` is used, and to send a START character, `TCION` is used. Generally, most terminals handle starting and stopping of the display from the device driver level and not from a user program; however the use of this routine is provided for compatibility with devices not controlled by the tty device driver.

RTS/CTS flow control

The Ready to Send (RTS) and Clear to Send (CTS) lines for the RS-232 serial interface can be used as handshaking signals between a computer and a modem. When the computer has data to send, the modem asserts CTS indicating that it is ready to receive data. The modem receives data until its buffer fills, then it lowers CTS and the computer stops sending data. When the modem can receive more data, it asserts CTS. This procedure repeats until all the data is received at the modem.

Similarly, when the modem has data to send, the computer asserts RTS indicating that the computer is ready to receive data. The modem then sends data until the buffer fills at the computer. Then the computer lowers RTS and the modem stops sending data. When the computer is ready to receive more data, it asserts RTS. Again, this procedure repeats until all data is received at the computer.

RTS/CTS flow control is enabled with the `stty ctsflow` and `rtsflow` options. More information is provided on the `stty(C)` and `termio(M)` manual pages.

In the `tty(q.v.)` structure, the `c_iflag` is set to `CTSFLOW` and `RTSFLOW` to enable this feature.

Flush

When a terminal session completes abnormally such as when the phone line carrier signal drops unexpectedly, any data not transmitted or received remaining in the terminal's read or write queues (buffers) is discarded. The act of discarding the data is known as flushing.

Non-STREAMS data flushing

Flushing is performed from a user program with the `tcflush(S)` routine. From a *device driver* (q.v.), the `ttyflush(K)` function is used. `tcflush(S)` simply calls `ttyflush(K)`. These routines are typically called by code cleaning up structures before a device is closed.

The queues that are flushed depend on whether a read or write operation is being performed. Also, both read and write queues can be flushed from the same routine call. When the read queues are flushed, the contents of the *tty structure* (q.v.) `t_canq` and `t_rawq` queues are discarded. When the write queue is flushed, the contents of the `t_outq` field are discarded.

The `tcflush(S)` format for the `tcflush` routine is:

```
int tcflush (filedes, queue_selector)
int filedes, queue_selector;
```

Set *queue_selector* to `TCIFLUSH` for flushing read queues, to `TCOFLUSH` for write queues, or `TCIOFLUSH` for both.

STREAMS data flushing

Under *STREAMS* (q.v.), flushing means removing messages from the message queue using the **flushq** routine. The following example shows message queue flushing using **flushq**:

STREAMS canonical flush handling

```

1      switch (mp->b_datap->db_type) {
2      . . .
3      case M_FLUSH:
4          /* Canonical flush handling */
5          if (*mp->b_rptr & FLUSHW) {
6              flushq(q, FLUSHDATA);
7              s = spl5();
8              /* also flush lp->msg because it is logically */
9              /* at the head of the write queue           */
10             if (lp->msg) {
11                 freemsg(lp->msg);
12                 lp->msg = NULL;
13             }
14             splx(s);
15         }
16         if (*mp->b_rptr & FLUSHR) {
17             flushq(RD(q), FLUSHDATA);
18             *mp->b_rptr &= ~FLUSHW;
19             qreply(q, mp);
20         } else
21             freemsg(mp);
22         break;

```

line 1 A message pointer type field passed to the routine is tested to determine the activity to be initiated.

lines 5 through 14

For write queue flushing of a character device, **flushq** is called to release the message queue and all message blocks. In line 7, all *interrupts* (q.v.) of level 1 to 5 are blocked with the set priority level (**spl**) routine. This prevents interrupts from other character devices and low-level system events from interrupting the next few lines of code until **splx(K)** is called to reset all previously enabled interrupts. In line 10, the message queue for the leading message is flushed, all the associated message blocks are released in line 11, and the pointer to the leading message is set to null in line 12. The code then drops through to the next section for flushing read queues.

lines 16 through 22

For flushing read queues, the message queue and all related message blocks are released in line 17. In line 18 the message pointer field is set to being not a write. Then the message pointer is sent in the reverse direction on the stream. In line 21, if the message pointer was not set for a read flush, the message blocks associated with the message pointer are cleared.

form.h header file

The `<form.h>` header file specifies constants, macros, structures and routines used by FORMS, an on-screen form generation package that is part of the *extended terminal interface* (q.v.) (ETI).

See also "Header files" in this *Encyclopedia* and *User Interfaces Guide* manual.

FORTRAN

The FORTRAN (FORMula TRANslator) language was one of the first high-level programming languages. It was developed in the late 1950s to make the description of algorithms easy and precise. ANSI has standardized three versions, commonly called FORTRAN-66, FORTRAN-77, and FORTRAN-90. FORTRAN remains popular for scientific and other applications that need its powerful mathematics capabilities.

FSS (File System Switch)

The File System Switch (FSS) facility allows the kernel to simultaneously support multiple file system architectures, such as network file systems and file systems associated with other operating systems. Processes use the normal UNIX system calls to access files, and the kernel maps a generic set of file operations into operations specific to each file system architecture. Since this is a feature of the kernel, a program can be written to access files without having to be concerned with the file system architecture.

For more information, see Maurice Bach's *Design of the UNIX Operating System* "File System Abstractions" (pages 138-139). For an early discussion of the principles behind FSS, see Killian, T.J., "Processes as Files," *Proceedings of the USENIX Conference, Summer 1984*, pp. 203-207. The Killian article should be used for background only; the SCO UNIX System V/386 implementation is somewhat different than that discussed, most notably in that the "process-as-file" abstraction is not supported.

ftw.h header file

The `<ftw.h>` header file specifies constants and routines used to walk a file tree.

See also "Header files" in this *Encyclopedia* and the manual page for `ftw(3)`.

Function (C language)

A C language function is a collection of declarations and statements that perform a specific task. Subroutines that users code can be called as functions from any program, assuming that the program has access to that subroutine. When writing a function, be sure that it is coded to handle error conditions correctly; see the “Error handling” article for more information.

When you issue a Section S function call, always provide for detection and handling of error returns. The manual pages give the expected return value for each function and define the meanings of error values that can be returned. See the “Error handling” article for more information.

Declaring functions

A function must be declared before it is used; otherwise, the compiler treats the function as returning an integer. In some cases, the program may compile, but it fails on execution and produces an error message and possibly a core dump.

Integer sizes vary depending on the memory model used and the machine architecture. If a function that returns a double is not declared, the compiler treats the return value as an `int`. In the example below, if `drand48` is undefined, the non-negative double-precision value returned fails, because only 16 or 32 bits have been allocated for storage. Declaring the function as a double before use is required to ensure proper handling.

Declaring a function

```

1      main()
2      {
3          double r, drand48();
4          int x;
5          long seed=1.0;
6          void srand48();
7          srand48(seed);
8          for (x=0; x<20; x++) {
9              r=drand48()
10             printf("Random number is %g\n",r);
11         }
12     }
```

Writing functions that take variable numbers of arguments

The macros documented on the `varargs(S)` manual page provide a means of writing portable functions that accept a variable number of arguments. The following program illustrates how this is done. All the arguments in the example, except the first, are character pointers, but they do not need to be. `My_Func` uses the first argument to determine how many arguments follow.

Function that takes variable number of arguments

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

main()
{
    My_Func(4, "hello", "world:", "insert", "coin");
}

My_Func(count, ...)
int count;
{
    va_list pa;
    char *p;
    va_start(pa);

    while (count-- > 0) {
        p=va_arg(pa, char*);
        printf("%s", p);
    }
    putchar('\n');
    va_end(pa);
}
```

Function prototypes

The second edition of Kernighan and Ritchie's *The C Programming Language* defines a function prototype as a declaration of a function, where "the types of parameters are explicit and are part of the type of the function." Using function prototyping enables the compiler to check that each function is getting the correct arguments. This check will catch a class of common programming errors.

Specific rules have been set up about prototypes and passing arguments to functions. If a function prototype does not appear in the code before it is used, nothing is assumed about the function arguments and so the compiler does not detect the error.

A contrived example of a function prototype is as follows:

Function prototype example

```
float fish(float x)
{
    return(x+1.0);
}
```

To understand how this prototype is used, consider the following code fragment that does not use a function prototype. The compiler does not detect that the programmer meant to pass **p* to *fish* rather than *p*.

Code example #1

```
main()
{
    /* error not detected */
    float y = 5.0, *p = &y;
    printf("The value is %f\n", fish(p));
}
```

Similarly, if a function declaration does not include arguments, the compiler makes no assumptions about these arguments. Again, the compiler does not detect that the wrong parameter was passed to *fish*.

Code example #2

```
main()
{
    /* error not detected */
    float fish(), y = 5.0, *p = &y;
    printf("The value is %f\n", fish(p));
}
```

The following code fragment includes the *float* prototype before calling *fish*. This example has a coding error, but it illustrates the proper use of prototyping.

Code example #3

```
main()
{
    /* error detected */
    float fish(float), y = 5.0, *p = &y;
    printf("The value is %f\n", fish(p));
}
```

Suppose "Code example #3" is compiled with the following command:

```
cc file.c -lm
```

The compiler produces the following warning messages:

```
file.c
file.c(5): error C2115; 'argument',: incompatible types
file.c(5): warning C4024: 'fish': different types: parameter 1
```

The programming tool *lint*(CP) will sometimes catch this kind of error.

The authoritative discussion of function prototypes is to be found in ANSI X3.159-1989 Programming Language — C.

Function (shell script)

In a Bourne or Korn shell script, you can declare a function that is used in the same manner as a subroutine in other programming languages. The function can be used with or without arguments. No additional declarations are required when arguments are used. The following example shows a function for a *.profile* file that creates a variant of the `cd` command that will change directory and place the current pathname in the command line prompt:

```
xd( )    {
        cd $1
        PS1=" 'pwd' > "
    }
```

In the example, when the `xd` function is executed the directory is changed to the argument to the `xd` function, and the `PS1` variable containing the command line prompt is set to the current pathname.

After placing the function in a *.profile* file, reinvoke the *.profile* file by entering:

`.$HOME/.profile`

Then use `xd` instead of `cd(C)` to change directories.

Arguments passed to shell functions are assigned to the shell variables `$1` to `$9`. The name of the function is assigned to `$0`. Refer to the "Argument" article for more information about how shell scripts capture and use arguments.

G

grap

grap is a preprocessor used with the **troff** text formatter to format graphs. The user embeds graph drawing commands in a **troff** document. The file is processed by **grap**, which converts these commands into the equivalent *pic* (q.v.) commands. The resulting file must be processed by **pic**, which produces **troff** input.

grap supports a small language for specifying graphs. The data that is plotted may be generated by the program, included in the same file as the program, or read in from an external file. **grap** can draw a frame for the graph and specify a coordinate system. Data can be plotted with solid or dotted lines, or as discrete points. Data points are drawn as dots by default, but can also be represented as bullets, stars, etc. The programming support provided by **grap** includes macros, loops, if-statements, and the ability to execute shell commands.

troff and **grap** are not included in the SCO UNIX System V/386 Development System, but are available from a number of outside vendors or as public domain software for UNIX, XENIX, and DOS systems. For more information about **troff**, refer to O'Reilly and Associates *UNIX Text Processing*.

Graphics adapters

Graphics adapters are circuit cards that display graphics, such dots, lines, curves, shapes, shading, and other imagery on a video monitor. A graphics adapter contains a processor and specialized programming for handling drawing, color generation and selection, alternate screen formats, two- and three-dimensional graphics, and animation. Graphics adapters are differentiated by the resolution, expressed in the number of pixels wide by the number of pixels high on a video screen, and the number of colors that can be displayed.

Some of the classes of graphics adapters are:

Adapter	Resolution	Colors
Super VGA	800x600	16 (color)
	1024x768	16 (color)
	1024x768	256 (color)
VGA	320x200	256 or 16 (color)
	640x200	2 (monochrome) or 16 (color)
	640x350	4 (monochrome) or 16 (color)
	640x480	2 (monochrome) or 16 (color)
EGA	320x200	16 (color)
	640x200	2 (monochrome) or 16 (color)
	640x350	4 (monochrome) or 16 (color)
CGA	300x200	4 (color)

The Color Graphics Adapter (CGA) was the first adapter developed for IBM PCs and compatible computers and has been largely superseded by the Enhanced Graphics Adapter (EGA) and the Video Graphics Array (VGA) adapters.

Graphics adapters have two display modes, text and graphics. In graphics mode all points on the screen are addressable. In text mode, the text is displayed using the default font or from downloadable fonts.

The `</usr/lib/grafinfo>` directory contains subdirectories for various manufacturers of graphics adapters. Each subdirectory contains files with a detailed description of a specific graphics adapter from the manufacturer. These files are accessed by programs that display information on the graphics adapter.

The console driver provides back-end support for developing video display adapter *device drivers* (q.v.) and is described in the *Device Driver Writer's Guide*. The console driver also supports `ioctl(S)` commands for performing a variety of functions. These commands are defined in the `<sys/vtkt.h>`, `<sys/machdep.h>`, and `<sys/console.h>` header files. The commonly used graphics adapter `ioctl` commands are described in the “`ioctl` routines” article.

The following example shows the use of four of these **ioctl** commands:

Graphics adapter ioctl example subroutines

```

1      #include <sys/machdep.h>
2      #include <sys/types.h>

3      int Adapter, Textmode;
4      char *Screenmem;

5      graphics_mode()
6      {
7          Adapter = ioctl(0, CONS_CURRENT, 0);
8          Textmode = ioctl(0, CONS_GET, 0);
9      }

10     text_mode()
11     {
12         ioctl(0, MODESWITCH | Textmode, 0);
13     }

14     mapframebuffer()
15     {
16         Screenmem = (char *) ioctl(0, MAPCONS, 0);
17     }

```

- | | |
|----------------|---|
| lines 1 and 2 | The header files are included. <i><sys/machdep.h></i> contains the definitions for the ioctl commands illustrated in these examples. |
| lines 5 to 9 | In this routine, the graphics adapter type is acquired in line 7 and then the text mode variety is saved off so that it can be restored in the text_mode routine in lines 10 to 13. |
| lines 10 to 13 | In this routine, the mode is switched back to text mode. This routine is generally called at the end of a program before exit. |
| lines 14 to 17 | In this last example routine, a request is sent to the kernel to map the video card's physical frame buffer into this process' address space. The virtual address that the user process can use is returned in the return value of the ioctl function. |

group file

The `/etc/group` file defines the groups recognized by the system for file access and use of other system resources such as the *IPC facilities* (q.v.). Groups can be assigned to correspond to projects, departments, or other sets of users who need to access a set of files and resources. Each file has a group, determined by the *EGID (effective GID)* (q.v.) of the process that created the file or assigned with the `chgrp(C)` command or `chgrp(S)` system call after the file is created.

Each group has an ID number, the *GID* (q.v.). The members of each group are listed in the right-most field of the `/etc/group` file; users can use the `newgrp(C)` command to change the effective GID to another group, as long as the user is listed as a member of that group in the `/etc/group` file. See the *System Administrator's Guide* for information about creating groups and assigning members to them.

Group file control routines

The *group* file control routines provide you with low-level control of the *group* file. Only the super user can modify these files, but any user can search them. These routines are discussed on the `getgrent(S)` manual page. (The *group* structure is defined in the `<grp.h>` header file.)

Routines to access `/etc/group` file

Routine	Description
<code>endgrent()</code>	close the group file
<code>struct group *getgrent()</code>	read the next line of the group file
<code>struct group *getgrgid(gid)</code>	search the group file from the beginning for a match to <i>gid</i>
<code>struct group *getgrname(name)</code>	search the group file from the beginning for a match to a <i>name</i>
<code>void setgrent()</code>	rewind the group file

grp.h header file

The `<grp.h>` header file defines the *group* structure containing pointers to information about entries in the `/etc/group` file. For more information, see the “group file” article in this *Encyclopedia* and the `getgrent(S)` manual page.

H

Hashing

“Hashing” is a searching technique based on mapping a large, sparse set of keys to a smaller, denser set of indices into a table. A typical example is its use in the *kernel* (q.v.) tables. One example is the table of in-memory disk buffers. Because many buffers are present in memory (frequently hundreds of them), rather than sequentially examining this very large table to find a particular block, the kernel creates an index using hashing, which is used to access the block quickly. Ideally, a more efficient way to find a block would be a simple table index by the disk block number, but because block numbers can be 32 bits, this would require more than four million entries. Using a hashing algorithm, the 32 bits are reduced to a smaller 8-bit value. All the items that map to a given number are kept in a list. Then that list (which is a small subset of all the entries) is searched to find the desired entry.

Header files

A header file is an ASCII file that contains C language constructs. UNIX systems use header files to localize information. Localizing common information enhances the portability of code and ensures that all programs compiled using the header file use the same definitions for specific entities. If information changes in the header file, all programs that reference that header file incorporate those changes when they are recompiled. Without the header files, all programs would have to declare these structures and then be recoded when the definitions change. The use of header files is a major help when porting programs from one system to another and when two or more programs must agree on the meaning of a defined variable.

For technical users, header files also provide a quick reference to data structures implemented on the system.

On UNIX systems, system header files are located under the */usr/include* directory. This directory contains a number of definitions of interest to user-level programs. For example, *<a.out.h>* and *<x.out.h>* define the structure of executable object files, *<values.h>* defines the sizes of various data elements, and *<curses.h>* defines the library used to generate formatted terminal screens. The */usr/include/sys* directory contains header files that define structures used by drivers and other kernel facilities, such as *<user.h>* and *<proc.h>* (which describe the basic data structures that define a user process). Header files in the */usr/include/sys* directory also define kernel macros and I/O controls (ioctl's)

that are used by user-level processes to control drivers. Use the `-I` option to the `cc`(CP) command to add other directories to these that are searched by the compiler; the `-X` option to `cc` specifies a different default path to be used for header files.

A program that needs to access definitions in a particular header file uses a `#include` statement referencing the file.

Details on specific header files appear in articles named for those header files.

High-level language

A high-level language is a language that can be used to write computer *programs* (q.v.) that are, theoretically, independent of the architecture or instruction set of the machine on which they are written. This means that a program written in a high-level language on one machine can be run, with few if any changes, on any other machine that supports that high-level language.

Initially computer programs were written using *binary* (q.v.) numbers that corresponded to specific machine instructions and memory locations on the computers for which they were written. This evolved into *assembly languages* (q.v.) which allowed the machine instructions and memory locations to be referred to by symbolic names. Language statements, made up from these symbolic names, were then translated back into the machine instructions and memory locations through the use of an *assembler* (q.v.). Assembly languages are considered low-level languages because the correspondence between language statement and machine-level instruction is still one to one. This prevents an assembly language program from being run on any machine that does not have the same instruction set.

High-level languages were developed to be independent of any particular machine instruction set. They take abstract concepts such as decision making and repetition, and define a language *syntax* (q.v.) that can be used to describe these concepts. Language statements using this syntax can then be translated into machine instructions through the use of a compiler. Often a single high-level language statement translates to several machine instructions, unlike the one to one correspondence of an assembly language statement. Any machine providing a compiler for a high-level language is able to use a program written in that language. This makes high-level languages independent of any particular architecture or machine instruction set. Some of the more commonly used high-level languages are C, *FORTRAN* (q.v.), *COBOL* (q.v.), Pascal, and Ada.

Host adapter

A host adapter is a device (generally a card) that converts computer-independent *SCSI* (q.v.) communications protocol to computer-specific information that your computer can process.

HyTime

The Hypermedia/Time-Based Structuring Language, also known as HyTime, is a standard language for describing properties of "hypermedia documents". HyTime is the subject of the ISO draft standard 10744.

Hypermedia documents are of two types: hypertext and multimedia. (The term "document" should be understood in a general sense here.) Hypertext is information that can be accessed in different orders; multimedia information can be used with different communications technologies. HyTime was devised to deal with the problem of interchanging and combining hypermedia documents of different formats.

HyTime is not a standard for representing data; rather it is used to co-ordinate hypermedia documents that may be in any format. HyTime constructs are used to specify linking, in particular, *hyperlinks*, which provide entry points into hypertext documents. HyTime can also specify time-scheduling and synchronization, which are especially important when using multimedia information. A processor called a *HyTime engine* understands these constructs, and can, for example, direct an application-specific program to carry out operations on the documents.

I

I18N

“I18N” is an abbreviation for *internationalization* (q.v.). The origin of this abbreviation is debatable, but it is commonly used within the computing industry.

ieeefp.h header file

The <ieeefp.h> header file specifies constants, macros, variables, structures and routines used to interface with floating-point coprocessor chips that meet the IEEE floating-point standard.

See also “Header file” in this *Encyclopedia*.

i_errno.h header file

The <nls/i_errno.h> header file specifies constants and variables used to provide error values for native language support, more recently referred to as *internationalization* (q.v.).

See also “Header files” in this *Encyclopedia* and the manual page for Intro(S).

Ilist

The ilist contains all the *inodes* (q.v.) for a *filesystem* (q.v.), arranged contiguously, starting at logical block number 2 of the file system. The *inumber* (q.v.) gives the starting point for the inode as an offset into the ilist.

INGRES

See “Structured Query Language (SQL)”

Inode

An inode contains the control information for a file. Each regular file, *pipe* (q.v.), special device or file, or directory has an inode associated with it; it contains all the information about the file except its name (kept in a directory data storage block) and its contents (kept in data storage blocks that are pointed to by the inode). Note that the inodes for special files do not point to data storage blocks, but instead store the *major and minor numbers* (q.v.) of the device. Inodes also are used with XENIX shared data and XENIX semaphores.

There are two types of inodes, on disk and in-core. An inode for a file contains a number of structure members, which are defined in the `<sys/inode.h>` header file for in-core inodes and in the `<sys/ino.h>` header file for on-disk inodes. The major information contained on disk in an on disk inode is as follows:

mode	file type (regular, directory, special, or pipe), execution bits (sticky bit, set user and group ID's), and access <i>permissions</i> (q.v.) for the owner, group, and others.
number of links	number of times this file is referenced in a directory, that is, the number of links to the file. Regular (non-named) pipes have a zero value in this field.
owner and group ID's	owner and group identification for the file. These ID's are used when checking access permissions for the file. Both of these are numeric values; the translation to character strings can be found in the <code>/etc/passwd</code> and <code>/etc/group</code> files.
size	number of bytes in the file. If the current offset in the file is the same size as that of the file, the user is at the end of the file. For a pipe file, this field shows the number of bytes written but not yet read. For a special device file, this field shows the major and minor numbers of the device.
access times	three times are kept: the last time the file was modified, the last time the file was accessed, and the last time the inode was changed. Simply reading a file does not affect the modification time or changed time, but does affect the access time. Writing to a file does not affect the access time but does affect the other two times.
data block addresses	an array of 13 data block numbers used to access the data storage blocks for the file

Inode table, system (in-core)

The system or in-core *inode* (q.v.) table is the *filesystem access table* (q.v.) that holds most of the information from the on-disk version of the inode, plus the following:

device	identifies the device on which the <i>filesystem</i> (q.v.) which contains this inode resides
inumber	with "device," locates the file's on-disk inode. The location is not stored in the on-disk version of the inode.
reference count	number of times the file is open; in other words, the number of pointers from the system file table to this entry. When this number drops to zero, the file is no longer being referenced and the entry can be deallocated.
last logical block read	used by the system to decide if the file is being read sequentially. If it is, the operating system invokes the read-ahead feature each time a read operation is done. This causes the block $n+1$ to be queued for reading when the block n is read, assuming that it will be needed eventually.
hash list pointers.	used to locate used inode table entries and maintain a free list of inode table entries
flags	indicate whether the file represents a shared-text <i>a.out</i> file (q.v.), a locked inode, a modified file, or a mount-point directory. Other flags specify if synchronous writes are to be done to this file, and if some process is waiting for this inode to be unlocked.

Note that the system inode table does not contain any time stamps, which are part of the on-disk version of the inode.

Several linked lists are used to provide easy access to system inode table entries currently in use. They also keep a list of free inode table entries for use when allocating entries while opening files. UNIX filesystems include the following inode table lists to decrease the time required to manipulate new files:

free list	starts at an operating system variable, which points to an unused system inode table entry, which points to another, and so on until all free table entries are on the free list. The link pointer of the last entry has a null pointer as its value.
hash lists	consist of pointers to inode table entries, each of which starts a hash list of inode table entries. This reduces the average number of entries inspected when you look for a file in the table. This feature separates the total number of used entries into a series of lists specified by the S5CACHEENTS kernel parameter. These lists are maintained through an inode hash table.

The inumber and device are hashed to find an entry in the hash table. This entry points to an entry in the system inode table, which is the head of the hash list. If the file is represented by an entry in the inode table, it is on this hash list.

The size of the system inode table is controlled by the kernel tunable parameter, **NINODE**.

Use the **inode** command to the **crash(ADM)** command to display the system inode table on your system. The following is an example of this display:

```

crash
> inode
INODE TABLE SIZE = 300
SLOT MAJ/MIN FS INUMB RCNT LINK  UID  GID  SIZE  MODE  MNT M/ST RCVD FLAGS
 19  1,40  1 1061  1  1  0  20    0 c---440  0 S  0  -  ac
 24  1,40  1 2979  1  5  2  2 131560 f--v711  0 S  0  -  ac tx
 71  1,40  1  319  2  1  2  20 153520 f---111  0 S  0  -  ac tx
152  1,40  1  407  1  1  2  2   864 f---664  0 S  0  -
176  1,40  1 3837  1  1  0  1    0 f---644  0 S  0  -
185  1,40  1  254  1  6  2  2  23980 f--v555  0 S  0  -  ac tx
282  1,40  1   27  2  2  2  2  52724 f--v711  0 S  0  -  tx
285  1,40  1 3105  2  9  2  2   160 d---644  0 S  0  -
291  1,40  1  124  1  3  2 15    0 c---600  0 S  0  -  up ac ch
295  1,40  1   72  4  1  2  2  47743 f---755  0 S  0  -  ac tx
297  1,40  1   58  2  2  2  2  31388 f---100  0 S  0  -  tx
299  1,40  1    2  5 14  0  2   544 d---755  0 S  0  -  up ch rt

```

To see detailed information on a particular inode, use **inode -f inumber**.

Intel binary compatibility specification (iBCS)

The *Intel386 Binary Compatibility Specification, Edition 2* (iBCSe2) defines a system interface for compiled application programs using the *System V Interface Definition, Issue 2* (AT&T SVID Issue 2) and the *Portable Operating Systems Interfaces, Part 1 (POSIX)* (q.v.) standards. The iBCSe2 standard applies to the Intel 80386 and 80486 processors, and includes machine instructions for the Intel 80386 processor and higher level library routines.

To produce an iBCSe2 compliant program, use only the system calls, commands, and other facilities described in the standards document and use the **cc -iBCS2** option to compile the program.

The following section (S) routines comply with the iBCSe2:

abs	fgets	ioctl	puts	strncat	t_rcvconnect
access	fopen	isatty	putw	strncmp	t_rcvdis
atof	fprintf	isnand	read	strchr	t_rcvrel
atoi	fputc	kill	realloc	t_accept	t_rcvudata
atol	fputs	ldexp	sbrk	t_alloc	t_rcvuderr
brk	fread	lseek	setbuf	t_bind	t_snd
calloc	free	malloc	sighold	t_close	t_snddis
cfree	freopen	memccpy	sigignore	t_errlist	t_sndrel
chdir	frexp	memchr	signal	t_errno	t_sndudata
chmod	fseek	memcmp	sigpause	t_error	t_sync
close	fstat	memcpy	sigrelse	t_free	t_unbind
creat	fwrite	memset	sigset	t_getinfo	time
ecvt	gcvt	mktemp	sprintf	t_getstate	ungetc
errno	getchar	open	stat	t_listen	unlink
fclose	getenv	optarg	strcat	t_look	utime
fcntl	getopt	opterr	strchr	t_nerr	write
fcvt	getpid	optind	strcmp	t_open	
fflush	gets	printf	strcpy	t_optmgmt	
fgetc	getw	putchar	strlen	t_rcv	

Internationalization

Internationalization is concerned with the design and implementation of software that adapts itself to a *locale* (q.v.). Global distribution of software packages presupposes that software designers and programmers can not assume that the English language or American customs are prevalent in all markets. This implies that an *application* (q.v.) must not rely on any particular language, character set, or set of cultural conventions. Thus, internationalized software is designed to apply to a generic location to which *localization* (q.v.) techniques can be applied to configure it to a particular region. To fully internationalize code, locale-specific issues should be decided through the use of pre-defined datafiles, pre-compiled message libraries, and various *environment* (q.v.) variables at runtime.

Some guidelines to follow when developing software with internationalization in mind are:

- Do not use the 8th bit of an *ASCII* (q.v.) character for purposes such as parity checking or guarding against sign-extension. International applications use the 8th bit to represent a full 256 element character set. The International Development System uses the ISO 8859/1 8-bit character set, also known as *Latin alphabet No. 1* (q.v.), rather than the ASCII set. This standard was chosen because it includes most western European characters.
- Do not assume 8 bits are enough to represent one character. Certain languages (most Asian languages, for example) require more than a single byte to represent a single character. This type of representation can be implemented by using either wide characters or *multibyte characters* (q.v.). The software should allow for these types of representations and should not restrict itself to single byte interpretations.

- Do not assume one character set over another. For operations such as case conversion or character collation, no assumption can be made about either the relationship between characters or the order of particular characters within a character set. Instead, write the software to read either the LC_COLLATE or the LC_CTYPE environment variables to get the order of characters for sorting operations and the relationship between characters for case conversion operations. See the `setlocale(S)` manual page or “locale” in this *Encyclopedia* for more information on the LC environment variables.
- Do not include or require the use of characters that may not be present on international keyboards. The characters in question are:

@ \$ () [] ~ ^ | \ , ' ,

- Do not assume a particular language will be used or understood by the user. Rather than hard-coding input, output, and error messages, use separate tables or *message catalogs* (q.v.) that are referenced outside of the code. Use the LC_MESSAGES and the NLSPATH environment variables (q.v.) to locate and use these message catalogs. See the `mestbl(M)`, `catget(S)`, and the `catopen(S)` manual pages for more information on message catalogs. For an example on how to construct and use a message catalog, refer to *Developer's Topics*.
- Do not assume a particular format for *date* (q.v.) and *time* (q.v.) reports. Instead, use the LC_TIME environment variable to get the proper format for date and time reports.
- Do not assume a particular punctuation for numeric and currency outputs. Instead, use the LC_NUMERIC and LC_MONETARY environment variables to get the proper format for numeric and currency outputs.

If this minimal set of guidelines are followed, the programmer will have accounted for most of the issues concerning internationalization. For further information on internationalization, see the OSF's *Internationalization Made Easy*.

Internet addresses

If you are using a communications network based on TCP/IP (q.v.), each machine connected to the network requires a unique Internet address. Even if you are not using TCP/IP for your network communications, if you want to use a top-level *domain* (q.v.) other than UUCP (UNIX To UNIX Copy) you need to have at least one Internet address issued by the DDN Network Information Center of SRI International, also known as the NIC (Network Information Center).

An Internet address is a 32-bit (4-byte) numeric value that specifies a particular network and a particular machine on that network. Internet addresses are specified in a standard format of four fields separated by dots, for example, 132.147.58.90. Each of the four fields represents one byte of the complete Internet address.

The Internet address can be specified in decimal, octal, or hexadecimal. Octal characters are preceded with a 0 (zero) and hexadecimal values are preceded with 0x or 0X. To allow flexibility in interconnecting multiple networks and multiple machines within a network, Internet addresses can use one, two, or three of the four fields to specify the network. The portion of the Internet address not used to specify the network is available to specify a particular host within that network.

Three classes of Internet addresses are provided depending on the quantity of machines that will be connected to the TCP/IP-based network. If your site anticipates connecting up to 256 machines, you are issued a Class C address. If you will connect up to 65,536 hosts, you are issued a Class B address, and if you anticipate a need for more than 65,536 hosts you are issued a Class A address. The class determines how the four bytes of the address are formatted as shown in the following table:

Internet Address Formats

Address Class	Network Portion	Host Portion	First Byte Value
A	Byte 1	Bytes 2 - 4	0-127
B	Bytes 1 and 2	Bytes 3 and 4	128-191
C	Bytes 1 - 3	Byte 4	192-223

The *Network Portion* of the address uniquely identifies your location on the network. The *Host Portion* of the address uniquely identifies a specific machine. The first byte of the address identifies the class type.

Interpreted language

An interpreted language is processed by a software system at runtime to simulate the execution. In contrast, a compiled language is translated into machine instructions which can be executed directly on the hardware.

Languages are commonly referred to as "interpreted languages", although, strictly speaking, interpretation is not a property of the language but rather of the implementation. Many languages have been implemented as both interpreters and compilers. *BASIC* (q.v.) and *LISP*, for example, are languages that are often referred to as interpreted even though compiler implementations exist for both.

There are both advantages and disadvantages to interpreted languages. One disadvantage is that an interpreted program usually runs more slowly than it would in compiled form because more machine code must be executed per instruction to perform the simulation. However, an interpreter often provides a better environment for debugging because the program can be run interactively; the program can be interrupted and the program state queried during execution which is much more difficult to do directly in a compiled program.

Interrupt

An interrupt is a notification from a device that tells the *kernel* (q.v.) that an action has been completed, that the device is ready to perform an action, or that the sending *process* (q.v.) or device requires immediate attention. The SCO UNIX System V/386 kernel depends on interrupts to schedule processing efficiently. Interrupts are processed by a programmable interrupt controller (PIC) on the mother board.

Intrinsics

The *X Window System* (q.v.) intrinsics are a set of user interface primitives free of visual and interaction style. The intrinsics provide a uniform way for developers to handle the common chores of *widget* (q.v.) construction. Among these chores are:

- initialization,
- input event dispatching (including enabling and disabling user input dispatch to sub-hierarchies of widgets),
- runtime configurability,
- uniform handling of common events (such as exposure and resize), and
- cleanup.

The intrinsics also include the uniform *application programming* (q.v.) interfaces for creating, controlling, and destroying widgets.

The full library of X Window System intrinsics is described in O'Reilly and Associates' *The X Toolkit Intrinsics Reference Manual, Volume 5*. See the *SCO Xsight Development System X Toolkit* for more information on Intrinsics and appropriate header files.

Inumber

The *inumber* provides a way to uniquely identify a file on a particular *filesystem* (q.v.). The *inumber* gives the index of the inode in to the inode list, referred to as the *ilist* (q.v.). To access the *inumber*, use either the *stat(S)* or *fstat(S)* system call or the *ls -i* command.

I/O subsystem

The I/O subsystem is the portion of the *kernel* (q.v.) that provides access to peripheral devices and pseudo-devices such as memory. The I/O subsystem consists primarily of *device drivers* (q.v.), which are the software interfaces between the devices and the operating system. It can be visualized as the layer of the kernel below the file subsystem and *process subsystem* (q.v.) and closest to the hardware itself.

User processes address devices through the *device special files* (q.v.) under the */dev* directory. Each special file contains a *major number* (q.v.) that the kernel uses to identify the driver that controls that device, and a *minor number* (q.v.) that the driver may use to identify the particular device or subdevice that is being accessed.

For a full description of the I/O subsystem, see Maurice Bach, *The Design of the UNIX Operating System*.

ioctl routines

An *ioctl* (I/O control command) routine is an entry-level point into a *device driver* (q.v.) that is accessed by user-level programs through the *ioctl*(S) system call. *ioctl* operations are used for operations other than opening, closing, reading, and writing the device. Typically, *ioctl* operations are used for device-specific activities such as controlling device hardware parameters, cleaning up character queues, and formatting disk devices.

ioctl calls can be used only with *character device drivers* (q.v.). A device that supports *ioctl* calls can be identified by an "i" in the second column of its entry in the */etc/conf/cf.d/mdevice* file, as shown below. Such devices will also have a "c" in the third column, to indicate that the device is a character device.

```
pa  ocIiwp  ictoHr  pa  0  6  1  3  -1
```

See the "Device drivers" article in this *Encyclopedia* and the *mdevice*(F) manual page for more information about the *mdevice* file.

Issuing the ioctl call from user programs

User-level programs use the *ioctl*(S) system call to initiate an *ioctl* operation:

```
ioctl(fildes, request, arg)
```

The device driver interprets the arguments *request* and *arg* from the open *file descriptor* (q.v.), *fildes*, that is associated with a *device special file* (q.v.). The *request* argument selects a control function that corresponds to this device. In most cases, the supported *ioctl* functions are defined in the header file for the device.

arg provides any additional information that the device needs to perform the control function specified by *request*. Depending on the request, the data type of *arg* is either an integer or a pointer to a device-specific data *structure* (q.v.).

Coding ioctl's in the driver

To create an *ioctl*:

- Define all commands in the driver's *header file* (q.v.) using all uppercase alphabetic strings.
- Code the driver's *ioctl* entry-point routine as a *switch* statement that contains *case* statements for each *ioctl* command that implement its functionality.
- Comment all routines thoroughly.
- Keep all *ioctl* values distinct from others on the system.

NOTE Although *ioctl* is the *system call* (q.v.) of choice for many I/O operations, XPG3 has withdrawn *ioctl* from its standard. Therefore, *ioctl* system calls cannot be used when pure XPG3 compliance is required.

Defined ioctl values

The following tables show some of the *ioctl*'s included in SCO UNIX System V/386 releases. The *ioctl*'s configured on your system may differ slightly from this list; check the header files before using any *ioctl* calls.

Terminal and tty ioctl's

The *ioctl* system calls for terminal files in the following table use this form:

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

ioctl's for terminal files (with termio structure)

Request	Description
TCGETA	gets parameters associated with the terminal and stores them in the <i>termio</i> structure referenced by <i>arg</i>
TCSETA	sets parameters associated with the terminal from the structure referenced by <i>arg</i> ; the change is immediate.
TCSETAW	waits for output to drain before setting the new parameters; use this form when you change parameters that affect output.
TCSETAF	waits for output to drain, then flushes the input queue and sets the new parameters

Other **ioctl** system calls for terminal files use this form:

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

The following routines use this form:

ioctl's for terminal files (without termio structure)

Request	Description
TCSBRK	waits for output to drain; if <i>arg</i> is 0, it sends a break (zero bits for 0.25 seconds)
TCXONC	starts and stops control; if <i>arg</i> is 0, it suspends output; if 1, it restarts suspended output
TCFLSH	if <i>arg</i> is 0, flushes the input queue; if 1, it flushes the output queue; if 2, it flushes both the input and output queues

For more information on terminal files, see the **termio(M)** manual page in the *User's Reference*.

The **ioctl** system calls for tty drivers in the following table use this form:

```
int ttioctl (tp, cmd, arg, mode)
struct tty *tp;
int cmd, arg, mode;
```

NOTE Unlike other **ioctl's**, the **ioctl's**, concerning tty drivers are accessible only from within the kernel; the **ioctl(S)** call from a process cannot be used.

ioctl's for tty drivers (with ttioctl structure)

Command	Description
LDOPEN	allocates a receive buffer; initializes <i>c_ptr</i> , <i>c_count</i> , and <i>c_size</i> ; and calls the driver's <i>xxproc</i> routine with T_INPUT as the argument
LDCLOSE	calls the driver's <i>xxproc</i> routine with the T_RESUME argument; calls <i>ttwait</i> to wait for data to clear the UART; and calls <i>ttyflush</i> to empty the <i>t_canq</i> and <i>t_rawq</i> buffers, as well as the contents of the <i>t_rbuf</i>
LDCHG	moves contents of the raw queue, <i>t_rawq</i> , to the canonical queue, <i>t_canq</i> (only works if ICANON is enabled)

Some **ioctl** system calls for tty drivers have the form:

```
int tticom (tp, cmd, arg, mode)
struct tty *tp;
int cmd, arg, mode;
```

The following commands use this form:

ioctl's for tty drivers (with *ttocom* structure)

Command	Description
IOC_SELECT	determines if a character can be read from or written to a tty device without going to sleep in the process. <i>mode</i> can be SELREAD or SELWRITE.
IOCTYPE	returns the name of the last ioctl called. <i>u.u_rval1</i> is set to the value of TIOC. IOCTYPE must not be called from an interrupt routine.
FIORDCHK	checks to see if characters are waiting to be read. 1 is returned if characters are waiting in <i>t_can</i> . If ICANON is set, it is also possible for 1 to be returned when characters are not in <i>t_can</i> , but there are characters in <i>t_delct</i> . If there are no characters in <i>t_can</i> and ICANON is not set, and if there are characters in <i>t_rawq</i> , 1 is returned.
XCSETAW†	waits for the UART to empty
XCSETAF†	waits until the UART empties, then flushes all read and write buffers
XCSETA†	sets terminal parameters from the tty structure specified by <i>arg</i> to <i>tticom</i>
XCGETA†	gets terminal parameters from the terminal's tty structure and puts into the tty structure specified by <i>arg</i> to <i>tticom</i>

- † Although these commands are AT&T SVID Issue 2 compliant, they are not, however, POSIX compliant. Use the **tcsetattr(S)** and the **tcgetattr(S)** routines instead for POSIX compliance.

For more information on interpreting tty driver **ioctl's**, see the **tticom(K)** manual page in the *Device Driver Writer's Guide*. For more information on tty driver routines, see the **tty(K)** manual page in the *Device Driver Writer's Guide*.

Defined AT&T *ioctl's*

For information about **ioctl's** that are included in any of the UNIX System V releases for the SBC and 3B computers, refer to the *AT&T Block and Character Interface (BCI) Driver Design Guide*.

Tape drives

Use the following `ioctl`'s to send commands to or query the status of a tape driver.

ioctl's for tape drives

Command	Description
MT_STATUS	returns a device-independent structure holding the status of the drive. The <code>tape_info</code> structure is defined in <code>/usr/include/sys/tape.h</code> . MT_STATUS can be used with all tape device drivers supported under UNIX systems
MT_RESET	resets the driver software and the tape drive; interrupts tape commands in process; can be used with all tape device drivers supported under UNIX systems
MT_REPORT	returns an integer code that determines the type of device that the driver controls. The type numbers are defined in <code>/usr/include/sys/tape.h</code> . MT_REPORT can be used with all tape device drivers supported under UNIX systems
MT_RETEN	winds the tape forward to the end of the tape and then backward to the beginning of the tape; can be used with all tape device drivers supported under UNIX systems
MT_REWIND	rewinds the tape to beginning of the tape; can be used with all tape device drivers supported under UNIX systems
MT_ERASE	erases the data on the tape and retensions the cartridge; can be used with QIC-02 cartridge tape drivers, SCSI tape drivers, and QIC-40 and QIC-80 mini-cartridge tape drivers
MT_AMOUNT	returns an integer count of the amount of the last data transfer; can be used with all tape device drivers supported under UNIX systems
MT_FORMAT	formats the tape; uses the number of tracks to format (must be an even number) as an argument; can be used on QIC-40 and QIC-80 mini-cartridge tape drivers and Irwin mini-cartridge tape drivers
MT_GETHDR	takes a pointer to a <code>struct ft_header</code> or <code>struct ir_header</code> as an argument and copies the header of the current tape into it; can be used on QIC-40 and QIC-80 mini-cartridge tape drivers and Irwin mini-cartridge tape drivers
MT_PUTHDR	takes a pointer to a <code>struct ft_header</code> or <code>struct ir_header</code> and writes it onto a tape; can be used on QIC-40 and QIC-80 mini-cartridge tape drivers and Irwin mini-cartridge tape drivers. Use this command with caution.
MT_GETNEWBB	takes a pointer to a <code>struct ft_newbbt</code> or <code>struct ir_newbbt</code> and copies in a list of bad blocks detected on the last write operation; can be used on QIC-40 and QIC-80 mini-cartridge tape drivers and Irwin mini-cartridge tape drivers
MT_PUTVTBL	takes a pointer to a <code>struct ft_btbl</code> and writes the volume table onto the tape; can be used on QIC-40 and QIC-80 mini-cartridge tape drivers only. Use this command with caution.

(Continued on next page)

ioctl's for tape drives (Continued)

Command	Description
MT_SERVO	writes servo marks on a blank tape in preparation for formatting with MT_FORMAT. If the tape has previously been servo-written, this command might fail unless the tape is first bulk-erased with a commercial tape eraser; can be used on Irwin mini-cartridge tape drivers
MT_RFM	winds the tape forward to the next file mark; can be used on QIC-02 cartridge tape drivers and SCSI tape drivers
MT_WFM	writes a file mark at the current location on the tape; can be used on QIC-02 cartridge tape drivers and SCSI tape drivers
MT_LOAD	loads the tape into the drive on devices that are capable of doing so; can be used only on SCSI tape drivers
MT_UNLOAD	unloads the tape from the drive on devices that are capable of doing so; can be used only on SCSI tape drivers

Console

Multiscreens can be reassigned to different adapters (in multi-adapter systems) with the following ioctl's.

ioctl's for console multiscreens

Command	Description
SWAPMONO	selects the monochrome display as the output device for the video monitor
SWAPCGA	selects the regular color display as the output device for the video monitor
SWAPEGA	selects the enhanced color display as the output device for the video monitor
SWAPVGA	selects the video graphics array color display as the output device for the video monitor

The following `ioctl`'s can be used to change the video display mode:

ioctl's for console display modes

Command	Description
SW_B80x25	selects 80x25 black-and-white text display mode (MONO, CGA, EGA, VGA)
SW_C80x25	selects 80x25 color text display mode (CGA, EGA, VGA)
SW_BG320	selects 320x200 black-and-white graphics display mode (CGA, EGA, VGA)
SW_CG320	selects 320x200 color graphics display mode (CGA, EGA, VGA)
SW_BG640	selects 640x200 black-and-white graphics display mode (CGA, EGA, VGA)
SW_EGAMONO80x25	selects EGA (Enhanced Graphics Adapter) mode 7; emulates support provided by the monochrome display (EGA, VGA)
SW_EGAMONOAPA	selects EGA support for 640x350 graphics display mode (EGA mode F) (EGA with mono monitor)
SW_ENHMONOAPA2	selects EGA mode F* (EGA with mono monitor)
SW_ENHB40x25	selects enhanced EGA support for 40x25 black-and-white text display mode (EGA, VGA)
SW_ENHC40x25	selects enhanced EGA support for the 40x25 color text display mode (EGA, VGA)
SW_ENHB80x25	selects enhanced EGA support for 80x25 black-and-white text display mode (EGA, VGA)
SW_ENHC80x25	selects enhanced EGA support for 80x25 color text display mode (EGA, VGA)
SW_CG320_D	selects EGA support for 320x200 graphics display mode (EGA mode D) (EGA, VGA)
SW_CG640_E	selects EGA support for 640x200 graphics display mode (EGA mode E) (EGA, VGA)
W_CG640x350	selects EGA support for 640x350 graphics display mode (EGA mode 10) (EGA, VGA)
SW_ENH_CG640	selects EGA mode 10* (EGA, VGA)
SW_MCAMODE	reinitializes the monochrome adapter (MONO)
SW_VGA40x25	selects VGA support for the 40x25 color text display mode (VGA mode 1+) (VGA)
SW_VGA80x25	selects VGA support for the 80x25 black-and-white text display mode (VGA mode 2+) (VGA)

(Continued on next page)

ioctl's for console display modes (Continued)

Command	Description
SW_VGAM80x25	selects VGA mode 7+; emulates support provided by the monochrome display (VGA with mono monitor)
SW_VGA11	selects VGA support for the 640x480 graphics display mode (VGA mode 11) (VGA)
SW_VGA12	selects VGA support for the 640x480 graphics display mode (VGA mode 12) (VGA)
SW_VGA13	selects VGA support for the 320x200 graphics display mode (VGA mode 13) (VGA)

NOTE Switching to an invalid display mode for a display device results in an error.

The following ioctl requests are provided to obtain information about the current display modes:

ioctl's for getting console display modes

Request	Description
CONS_GET	returns the current display mode setting for current display adapter (All)
CGA_GET	returns the current display mode setting of the color graphics adapter (CGA only)
EGA_GET	returns the current display mode setting of the enhanced graphics adapter (EGA only)
MCA_GET	returns the current display mode setting of the monochrome adapter (MONO only)
VGA_GET	returns the current display mode of the video graphics array adapter (VGA only)

The following **ioctl** commands can be used to map the display memory:

ioctl's for console memory mapping

Command	Description
MAPCONS	maps the display memory of the adaptor currently being used into the user's data space (All)
MAPMONO	maps the monochrome adapter's display memory into the user's data space (MONO only)
MAPCGA	maps the color adapter's display memory into the user's data space (CGA only)
MAPEGA	maps the enhanced graphics adapter's display memory into the user's data space (EGA only)
MAPVGA	maps the video graphics array adapter display memory into the user's data space (VGA only)

You can use the following **ioctl's** to input or output a byte on the graphics adapter port:

ioctl's for graphics adapter port I/O

Command	Description
MGAIO	inputs or outputs a byte on the monochrome adapter port as specified (MONO only)
CGAIO	inputs or outputs a byte on the color graphics adapter port as specified (CGA only)
EGAIO	inputs or outputs a byte on the enhanced graphics adapter port as specified (EGA only)
VGAIO	inputs or outputs a byte on the video graphics array adapter port as specified (VGA only)
CONSIO	inputs or outputs a byte on the current graphics adapter port as specified (All)

ioctl's for color console files

Request	Description
GIO_COLOR	returns information about console attributes, such as background color
GIO_ATTR	returns information about present character mode

Keyboards

ioctl's can be used to define or obtain the current definition of a console function key. The *arg* parameter of the **ioctl** call uses the **fkeyarg** data structure:

```
struct fkeyarg {
    unsigned int keynum;
    char keydef [MAXFK];
    /*Comes from char flen; ioctl.h via comcrt.h */
}
```

You can use the following **ioctl** commands to obtain or assign function key definitions:

ioctl's for function keys

Command	Description
GETFKEY	obtains the current definition of a function key; the function key number must be passed in keynum ; the string currently assigned to the key is returned in keydef and the length of the string is returned in flen
SETFKEY	assigns a given string to a function key; the function key number and the length of the string (number of characters) must be passed in flen

You can map keyboards using the following **ioctl's**:

ioctl's for keyboard mapping

Command	Description
GIO_KEYMAP	returns the current map
PIO_KEYMAP	puts the prepared buffer into place

You can map string keys using **ioctl's**. The syntax is:

```
#include <sys/keyboard.h>
ioctl(fd,cmd,buf)
int fd, cmd;
char *buf;
```

The following commands use this syntax:

ioctl's for mapping string keys

Command	Description
GIO_STRMAP	returns the string mapping table
PIO_STRMAP	puts the new string mapping table in place

The following **ioctl** commands are applicable to all STREAMS files:

ioctl's for STREAMS

Command	Description
I_PUSH	pushes the module whose name is pointed to by <i>arg</i> onto the top of the current <i>stream</i> , just below the <i>stream head</i> , then calls the open routine of the newly pushed module
I_POP	removes the module just below the <i>stream head</i> of the <i>stream</i> pointed to by <i>fildev</i> ; <i>arg</i> should be 0
I_LOOK	retrieves the name of the module just below the <i>stream head</i> of the <i>stream</i> pointed to by <i>fildev</i> , and places it in a null terminated character string pointed at by <i>arg</i> ; the buffer pointed to by <i>arg</i> should be at least FMNameSZ+1 bytes long
I_FLUSH	flushes all input and/or output queues, depending on the value of <i>arg</i>
I_SETSIG	informs the <i>stream head</i> that the user wants the kernel to issue the SIGPOLL signal when a particular event has occurred on the <i>stream</i> associated with <i>fildev</i> ; supports an asynchronous processing capability in STREAMS
I_GETSIG	returns the events for which the calling process is currently registered to be sent a SIGPOLL signal; returns events as a bit-mask pointed to by <i>arg</i>
I_FIND	compares the names of all modules currently present in the <i>stream</i> to the name pointed to by <i>arg</i> ; returns 1 if the named module is present in the <i>stream</i> ; returns 0 if the named module is not present
I_PEEK	allows a user to retrieve information in the first message on the <i>stream head</i> read queue without taking the message off the queue
I_SRDOPT	sets the read mode using the value of <i>arg</i>
I_GRDOPT	returns the current read mode setting in an <i>int</i> pointed to by the <i>arg</i>
I_NREAD	counts the number of data bytes in data blocks in the first message on the <i>stream head</i> read queue, and places this value in the location pointed to by <i>arg</i> ; the return value is the number of messages on the <i>stream head</i> read queue
I_FDINSERT	creates a message from user-specified buffers, addresses information about another <i>stream</i> , and sends the message downstream. The message contains a control part and an optional data part. The data and control parts being sent are distinguished by placement in separate buffers.
I_STR	constructs an internal STREAMS ioctl message from the data pointed to by <i>arg</i> and sends that message downstream; sends user ioctl requests to downstream modules and drivers
I_SENDFD	requests the <i>stream</i> associated with <i>fildev</i> to send a message, containing a file pointer, to the <i>stream head</i> at the other end of a <i>stream</i> pipe. The file pointer corresponds to <i>arg</i> , which must be an integer file descriptor.

(Continued on next page)

ioctl's for STREAMS (Continued)

Command	Description
I_RECVFD	retrieves the file descriptor associated with the message sent by an I_SENDFD over a <i>stream</i> pipe; <i>arg</i> is a pointer to a data buffer large enough to hold an <i>strrecvfd</i> data structure
I_LINK	connects two <i>streams</i> , where <i>fildes</i> is the file descriptor of the <i>stream</i> connected to the multiplexing driver, and <i>arg</i> is the file descriptor of the <i>stream</i> connected to another driver. The <i>stream</i> designated by <i>arg</i> gets connected below the multiplexing driver; requires the multiplexing driver to send an acknowledgment message to the <i>stream head</i> regarding the linking operation
I_UNLINK	disconnects the two <i>streams</i> specified by <i>fildes</i> and <i>arg</i> ; <i>fildes</i> is the file descriptor of the <i>stream</i> connected to the multiplexing driver; <i>fildes</i> must correspond to the <i>stream</i> on which I_LINK was issued; <i>arg</i> is the multiplexer ID number that is returned by I_LINK

IPC facilities

UNIX system *shared memory* (q.v.), *semaphores* (q.v.), and *message* (q.v.) queues are called IPC (Inter Process Communication) facilities because they were sold as add-on driver packages in older releases of UNIX Operating System V.

Each IPC mechanism uses a set of similar *system calls* (q.v.) to allocate and control the facility. These are described in the relevant sections later in this article. In this section, "entity" is a generic reference to a shared memory region, message queue, or semaphore array.

The general approach to using IPC mechanisms is similar for all facilities:

1. Programs that use IPC mechanisms must include the appropriate header files. These header files are located in the */usr/include/sys* directory:
 - `ipc.h` defines variables and flags used by shared memory, semaphores, and messages. All programs that use any IPC facilities must include this file.
 - `shm.h` defines structures used by the shared memory mechanism
 - `sem.h` defines structures used by the semaphore mechanisms
 - `msg.h` defines structures used by the messages mechanism
2. One process issues the `*get` (`msgget`, `semget`, or `shmget`) call to allocate the IPC entity or entities and associated data structures.¹ This `*get` call also

1. Many characteristics of the IPC data structures are controlled by tunable parameters that are explained in the "Messages, IPC," "Semaphores, System V," and "Shared memory, System V" articles in this *Encyclopedia*.

- determines file access *permissions* (q.v.) for the IPC entity, which control access as much as file access permissions control access to files.
3. All other processes access the IPC entity by its ID, and must issue a `*get` call that does not initialize any structures to get the ID of the IPC entity, typically using an appropriate *key* to identify the IPC entity.
 4. Processes then use the IPC facility through the operations described on the `*op` (`msgop`, `semop`, or `shmop`) manual page in section (S).
 - For shared memory, the operations allow the process to attach the shared memory region to the process so it can be read and/or written, and to detach the shared memory from the process if it is no longer needed.
 - For messages, the operations allow the process to send and/or receive messages from the queue.
 - For semaphores, the operations allow the process to decrement and increment the semaphore.
 5. If necessary, the `ctl*` calls (`msgctl`, `semctl`, and `shmctl`) allow any process with appropriate permissions to check and change the values of the controlling data structures. The `*ctl` command is sometimes used with the `IPC_RMID` command in the process that creates the IPC entity, to ensure that the memory associated with that entity is freed if the process exits or no longer needs the IPC entity. See the “Removing IPC Structures” section later in this article for more information.
 6. The `ipcs(ADM)` command can be used to get the status of any IPC entity being queried.
 7. The `ipcrm(ADM)` command can be used to remove a message queue, semaphore set, or shared memory ID from the system. This command should not be necessary if the `IPC_RMID` control command was issued (see step 5), but does allow the administrator to recover the memory if the code omits this call.

Each IPC mechanism has an associated structure that defines the individual shared memory segment, semaphore, or message. These structures are described on the `msg(FP)`, `sem(FP)`, and `shm(FP)` manual pages.

IPC keys

All IPC facilities require that processes supply a *key* (a long integer) to get the appropriate ID for the specific entity being accessed. This key can be any value that all processes agree on. If some standard is not used, unrelated processes may unintentionally access each others IPC entities.

The recommended approach is to use the `ftok(S)` routine to form a unique key from a filename and an ID that uniquely identifies a project or application. The file used as the first argument must correspond to an existing file on the system, such as the name of the executable file for the process that creates the IPC entity. Some programmers prefer to use different keys for different IPC types. For instance, the *key* for the shared memory region and the *key* for the semaphore array that protects access to the shared memory would be

different. However, the same key can be used, because the keys for each type of IPC mechanism are stored separately. See the manual page for more information about using `ftok`.

IPC flags

Each kind of IPC entity has a system call used to allocate the structure (`shmget(S)` for shared memory; `semget(S)` for semaphores; and `msgget(S)` for messages). Each of these calls takes a *flag* argument that determines the access permissions and operation mode for the IPC entity. The low-order nine bits of *flag* represent the desired permissions. Other bits select various options. Symbols (such as `IPC_CREAT`) for these bits are defined in `<sys/ipc.h>`. See example below.

Note that each IPC data structure has access permissions that are similar to those used on files (see the “Permissions, file access” article); read and/or write permissions are allocated to the owner, group, and/or other users. When the structure is created, its owner is the *EUID* (*effective UID*) (q.v.) of the creating process and its group is the *EGID* (*effective GID*) (q.v.) of the creating process. The UID and GID of the structure can be changed after it is created using the `shmctl(S)`, `semctl(S)`, or `msgctl(S)` calls. Access permissions must be defined or only the super user (who bypasses all permission controls) can access the structure.

The typical usage of the *flag* argument is to use numbers to represent the permissions and the predefined constants for other options. For example, the third argument to `shmget` is `shmflg`. To create a shared memory segment that has read and write permissions for the owner and group, the call is:

```
shmget(key, size, 0660 | IPC_CREAT)
```

Note that the execute permission bits are ignored for IPC entities. In other words, “0770” permissions for an IPC entity are exactly the same as “0660” permissions.

IPC flags

	Meaning	Predefined constant	Octal value
<i>Access permissions</i>	read by owner	SHM_R MSG_R SEM_R	0 0 0 0 4 0 0
	write by owner	none	0 0 0 0 2 0 0
	read by group		0 0 0 0 0 4 0
	write by group		0 0 0 0 0 2 0
	read by other		0 0 0 0 0 0 4
	write by other		0 0 0 0 0 0 2
<i>Operation (general)</i>	create entry	IPC_CREAT	0 0 0 1 0 0 0
	exclusive access (fail if <i>key</i> exists)	IPC_EXCL	0 0 0 2 0 0 0
	do not block. For semaphores, do not wait if operation can not be performed; for messages, do not wait if the queue is empty or full.	IPC_NOWAIT	0 0 0 4 0 0 0
<i>Operation (shared memory)</i>	attach with read-only permissions round attach address to SHMLBA	SHM_RDONLY SHM_RND	0 0 1 0 0 0 0 0 0 2 0 0 0 0
<i>Operation (messages)</i>	do not generate error if message is larger than MSGMAX	MSG_NOERROR	0 0 1 0 0 0 0

Removing IPC structures

Each shared memory region, message queue, and semaphore occupies a portion of system memory and is not automatically removed from memory when the process exits (either normally or abnormally). When you design applications that allocate IPC structures, you must decide whether these structures should be removed when the process exits:

- If the structure is not removed, the memory used by the structure is not available to other processes on the system. If the process is restarted and aborts several times, a large portion of memory might be wasted, which degrades general system performance.

If the structure should be destroyed when a process exits, code the **IPC_RMID** command with the **semctl**, **shmctl**, or **msgctl** call. This code is typically located in a subroutine that is called when the process exits.

- If the program executes continuously when the system is up and is coded to access the existing IPC structures if it is restarted, it can avoid the overhead involved in allocating memory for these structures by not removing them when the process exits.

The administrator also can use the `ipcrm(C)` command to remove orphaned IPC structures from the system. To use this, you must specify either the ID or the key of the entity. Use the `ipcs(C)` command to get the ID or key. See the manual pages for details about using these commands.

Checking the status of an IPC structure

The `shmctl(S)`, `msgctl(S)`, and `semctl(S)` calls with the `IPC_STAT` command as the second argument are used to check the current status of a shared memory segment, message queue, or semaphore array, respectively. They do that by copying the current value of each member of the controlling structure into a user level structure provided by the caller. The members of the structure are defined on the `msg(FP)`, `sem(FP)`, or `shm(FP)` manual pages. Note that the IPC structures include a structure that shows access permissions for the entity, so these values are also available through `IPC_STAT`.

For instance, the following call writes the current value of each member of the `myshm_ds` structure for a shared memory segment into the `check_shm` structure, then prints some of the information from `check_shm`:

```
shmctl(myshm_ds, IPC_STAT, &check_shm);
printf ("\nUser ID = %d\n", check_shm.shm_perm.uid);
printf ("\nThe number of processes attached = %d\n", check_shm.shm_nattch);
```

A similar procedure checks the status of a message queue or semaphore array. Note that `semctl` supports additional commands that allow you to check the value of an individual semaphore.

Changing access to an IPC structure

Each IPC entity has access permissions similar to those of files. In other words, it has an owner and group, with read and write permissions specified for the owner and the group. These permissions are determined when the entity is first created, but can be changed by a process that has the same effective user ID as the entity or a process with super-user privileges. To change the access permissions:

1. Make a copy of the existing control structure. For example:

```
rtrn = shmctl(myshm_ds, IPC_STAT, &temp_struct);
```

2. Modify the values in this structure. For example, the following code changes the UID to 306, the GID to 20, gives read/write privileges to the owner, and read-only privileges to the group:

```
temp_struct.shm_perm.uid = 306
temp_struct.shm_perm.gid = 20
temp_struct.shm_perm.mode = 0640
```

3. Replace the controlling structure with the new structure:

```
rtrn = shmctl(myshm_ds, IPC_SET, &temp_struct);
```

ISAM files

ISAM (Indexed Sequential Access Method) is a storage and retrieval system that allows programmers to manipulate files by record and/or field. SCO ISAM provides a library of C routines that enables a programmer to build custom data management procedures into an *application* (q.v.). It gives the programmer precise control over how data is handled, and so can be used when the commercial database management systems (DBMS) do not provide the appropriate data handling for the application or when the data handling portion of the application is too small to warrant the resources required by a full-scale database management system.

The SCO ISAM package is based on the X/Open standard with SCO developed enhancements, including:

- Case sensitivity may be turned off in string searches.
- Routines to generate statistics on the use of ISAM files are provided.
- Alternate, international collating sequences are available.
- Functions are provided to generate unique data values.

Keep these guidelines in mind when using ISAM files in your application:

- Include the *isam.h* header file (q.v.) in any code that uses ISAM routines.
- Use `cc -lisam` when compiling programs that use the ISAM routines.
- Before you run a program that accesses ISAM files, you must set the DBKEY environmental variable to some positive integer common to all users. For example, to set up DBKEY with a value of 100 using the Bourne Shell, type the command `DBKEY=100; export DBKEY` or to set up DBKEY using the C Shell, type the command `setenv DBKEY 100`.

The following table lists the routines used for ISAM files.

ISAM routines

Routine	Description
<code>isbuild(filename, recordlength, keydes, mode)</code>	creates a new ISAM file
<code>iserase(filename)</code>	deletes the specified ISAM file
<code>isrename(oldname, newname)</code>	renames a specified ISAM file
<code>isaddindex(isfd, keydesc)</code>	adds a new index to an existing ISAM file
<code>isdelindex(isfd, keydesc)</code>	deletes specified index from the ISAM file
<code>isopen(filename, mode)</code>	opens an ISAM file in specified access and lock modes
<code>isclose(isfd)</code>	closes an ISAM file

(Continued on next page)

ISAM routines (Continued)

Routine	Description
isstart(isfd, keydesc, length, record, mode)	selects an index and locates a specified record
isread(isfd, record, mode)	reads a record specified by the direction parameter
iswrite(isfd, record)	writes a record
iswrcurr(isfd, record)	writes a record and sets record position and record number
isrewrite(isfd, record)	rewrites record specified by the current key
isrewcurr(isfd, record)	rewrites current record and sets record number
isrewrec(isfd, recnum, record)	rewrites record specified by record number
isdelete(isfd, record)	deletes a record specified by the current key
isdelcurr(isfd)	deletes the current record
isdelrec(isfd, recnum)	deletes the record specified by record number
islock(isfd)	locks an ISAM file
isunlock(isfd)	unlocks an ISAM file
isrelease(isfd)	unlocks all locked records
isindexinfo(isfd, buffer, number)	gets information about an ISAM file and its keys
isuniqueid(isfd, uniqueid)	generates a unique data value
issetunique(isfd, uniqueid)	generates a unique incremented data value

Applications that use ISAM files will not execute on a UNIX System V/386 Operating System runtime system on which the SCO ISAM RTS is not installed. All current versions of SCO UNIX System V/386 include the ISAM RTS; it can be purchased separately for older releases of the operating system.

Systems on which ISAM applications run must be tuned to support the shared memory region and set of *semaphores* (q.v.) used by the ISAM routines. Applications should be coded to return meaningful error messages if these resources are not available. See the chapter on "Tuning System Performance" in the *SCO UNIX System V/386 Operating System, System Administrator's Guide* for more information on how to tune the *kernel* (q.v.) to support shared memory and semaphores.

See also the appropriate manual pages and the "Using ISAM" chapter in the *Developer's Topics*.

isam.h header file

The *<isam.h>* header file specifies constants, macros and external declarations for programs using the (ISAM) (q.v.) library.

See also "Header file" in this *Encyclopedia*.

ISO (International Organization for Standardization)

The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) based in Geneva, Switzerland, form a worldwide standards body that publishes a series of standards on electronics technology. Through national member organizations, technical committees meet to review and standardize technical activity. The committee for informational technology is ISO/IEC JTC 1. Draft standards adopted by the joint technical committee (JTC) are circulated to the national organizations before being accepted as International Standards.

UNIX operating system ISO/IEC standards

The following are some of the standards that apply to the UNIX operating system:

Standard	Description
ISO 3166	Country codes for MMDF domain names
ISO/IEC 9945-1	POSIX standard
ISO 8859/1	8 bit character set comprising most of the Western European characters
ISO 8072	Transport layer of Open Systems Interconnection (See also "OSI" article in this <i>Encyclopedia</i>)
ISO/IEC 9899:1990	Programming Language - C

J

Job Control

Job control is a feature of *Korn shell*, *ksh* (q.v.), which allows you to manipulate jobs or processes. Many implementations of *C shell*, *csh* (q.v.) also have job control; the version supplied with SCO UNIX System V/386 does not have this capability.

When a job is running in the foreground, the shell waits for the job to finish executing before displaying the prompt. When a job is in the background, the prompt is displayed and new command lines can be entered even though the job is still executing. With job control, you can stop and restart programs, and move them between the foreground and the background.

By default, jobs execute in the foreground. You can run a command in the background by adding an ampersand (&) character to the end of the command. With certain **stty(C)** settings, a foreground job may be stopped by entering (Ctrl)-Z; when a job is stopped, its execution is suspended but it is not killed. Stopped jobs and background jobs may be run in the foreground by typing the command **fg** at the prompt. Stopped jobs may be run in the background by typing the command **bg** at the prompt. The **jobs** command prints a list of all stopped and background jobs.

See the **ksh(C) manual page** (q.v.) and the User's Guide for information on writing programs that are compatible with **ksh** job control. See the **stty(C) manual page** for information on terminal settings. Refer to the *Developer's Topics* for information on using job control under the **ksh**.

K

Kernel

The kernel is the UNIX operating system and resides in the file */unix*. When booted, the kernel partitions the virtual address space into two sections: the user address space and the kernel address space. The user address space, is set to 2 gigabytes of address space (plus some space for the maximum amount of *shared memory* (q.v.)) and the kernel address space is set to approximately 1.5 gigabytes.

The kernel is an *a.out file* (q.v.) with the same format as a compiled C program. The text section of the */unix* file contains all of the *device drivers* (q.v.), system calls, and any other routines that comprise the kernel.

The kernel consists of three logical subsystems: I/O, process, and file. See also "Subsystems" in this *Encyclopedia*.

Korn shell, ksh

The Korn shell, *ksh*(C), is an interactive command-language interpreter and programming language that reads and executes *commands* (q.v.) from either the terminal or a *file* (q.v.). The *ksh*(C) combines features of the two most common UNIX shells, Bourne shell, and the C shell. The *ksh*(C) provides both compatibility with *sh*(C) and the command history and substitution features of *csh*(C). In addition, the *ksh*(C) includes command-line editing, job control, and enhanced command-history functionality.

See the *ksh*(C) manual page and Bolsky and Korn's *The Korn Shell Command and Programming Language*. For information about using job control under the *ksh*(C), refer to the *Developer's Topics*. See also "Job Control" in this *Encyclopedia*.

L

LANG environment variable

The *LANG environment* (q.v.) variable is one of the mechanisms that the user may choose to set *locale-dependent* (q.v.) values. *LANG* can be used to apply *localization* (q.v.) to software *applications* (q.v.) that implement *internationalization* (q.v.) techniques. Such considerations as the local language, the codeset in use, *date* (q.v.) and *time* (q.v.) formats, and monetary and numeric symbols can be resolved by setting the *LANG* variable to the conventions of a particular *locale* (q.v.).

The *LANG* variable can be placed in the */etc/default/lang* file or in the user's *.login* or *.profile* file or it can be set at any time within the *shell* (q.v.) for that current shell *process* (q.v.) (see the *environ*(M) manual page for the proper format for defining environment variables). The format for the values of *LANG* follows the locale naming convention:

`LANG=language_territory.codeset`

(see the *setlocale*(S) and the *locale*(M) manual pages for more information on possible values for *LANG*).

Other locale variables

There are other variables similar to *LANG* that also govern the behavior of locale-dependant settings. Using *setlocale*(LC_ALL ...) has an effect similar to setting the *LANG* variable. They are complementary because *LANG* is restricted to usage within a shell environment only while *LC_ALL* is restricted to usage within code only. Individual locale categories can be set independently using the optional environment variables of the same name:

`LC_COLLATE`
`LC_CTYPE`
`LC_MESSAGES`
`LC_MONETARY`
`LC_NUMERIC`
`LC_TIME`

Unlike *LANG* and *LC_ALL*, these category-specific environment variables can be set and used within the shell and/or within programs.

Customizing your environment

Through the use of these various locale variables, it is possible to combine the attributes of different locales to suit a user's specific needs. For example, a German character set can be used with time/date and numeric displays consistent with that of the United States by setting `LANG` to a standard German settings and then setting just `LC_TIME` and `LC_NUMERIC` to United States setting (Note that `LC_ALL` cannot be used instead of `LANG` because the setting for `LC_ALL` overrides all other locale variables. See "locale" in this *Encyclopedia* for details on the precedence rules for these variables).

Similarly, it is possible for users to define an environment that is independent of the default system value. For example, consider the following two lines, the first of which is found in `/etc/default/lang` and the second in the user's `.profile` file:

```
LANG=french_france.8859
```

```
LC_TIME=english_us.ascii
```

The default would become the French language, but with time/date conventions (including days of the week and months of the year) consistent with those used in the United States.

langinfo.h header file

The `<langinfo.h>` header file specifies constants and routines used by section (S) native language routines, those beginning with the `nl_` prefix. Native language support is provided for *internationalization* (q.v.).

See also "Header file" and "locale.h header file" in this *Encyclopedia*.

Latin alphabet No. 1

The SCO International Development System uses the ISO (*International Organization for Standardization*) (q.v.) 8859/1 8-bit single-byte coded graphic character set known as Latin alphabet No. 1, or simply Latin-1. The ISO 8859/1 character set includes characters from most Western European languages. It consists of the space character, 186 characters that form a subset of the of the graphic characters in ISO 6937/2 (1983), and four additional characters that are intended for inclusion in ISO 6937/2.

The standard specifically states that *control characters* (q.v.) are beyond the scope of ISO 8859/1. In practice, reference to this character set includes reference to the control characters that are specified in ISO 646, ISO 6429, and the *ASCII* (q.v.) character sets.

ASCII codes are represented with only 7 bits and so can represent 128 characters; the ISO 8-bit coded character sets specify an encoding scheme which uses 8 bits and can represent 256 characters. The tables that follow include the upper 128 characters. The ASCII tables include all of the graphic characters that occupy the lower 128 character positions of ISO 8859/1. Not all of the codes are currently assigned; in the tables that follow, a ☒ indicates an unassigned character.

Your ability to use this character set is governed by the capabilities of your data display and print devices and the presence of an appropriate character mapping file in `/usr/lib/mapchan`. Mapping files are provided for commonly used character mapping schemes; for detailed understanding of character mapping files, see `mapchan(F)`.



Octal values of 8-bit character set											
ISO-6429 Control Characters				ISO-8859/1 Graphic Characters							
200	␣	220	dcs	240	nbsp	260	°	300	À	320	Ð
201	␣	221	pu1	241		261	±	301	Á	321	Ñ
202	␣	222	pu2	242	¢	262	²	302	Â	322	Ò
203	␣	223	sts	243	£	263	³	303	Ã	323	Ó
204	ind	224	cch	244	¤	264	´	304	Ä	324	Ô
205	nel	225	mw	245	¥	265	µ	305	Å	325	Õ
206	ssa	226	spa	246		266	¶	306	Æ	326	Ö
207	esa	227	epa	247	§	267	·	307	Ç	327	×
210	hts	230	␣	250	¨	270	ˆ	310	È	330	Ø
211	htj	231	␣	251	©	271	˙	311	É	331	Ù
212	vts	232	␣	252	ª	272	°	312	Ê	332	Ú
213	pld	233	csi	253	«	273	»	313	Ë	333	Û
214	plu	234	st	254	¬	274	¼	314	Ì	334	Ü
215	ri	235	osc	255	shy	275	½	315	Í	335	Ý
216	ss2	236	pm	256	®	276	¾	316	Î	336	Þ
217	ss3	237	apc	257	¯	277	¿	317	Ï	337	ß

Hexadecimal values of 8-bit character set											
ISO-6429 Control Characters				ISO-8859/1 Graphic Characters							
80	␣	90	dcs	a0	nbsp	b0	°	c0	À	d0	Ð
81	␣	91	pu1	a1		b1	±	c1	Á	d1	Ñ
82	␣	92	pu2	a2	¢	b2	²	c2	Â	d2	Ò
83	␣	93	sts	a3	£	b3	³	c3	Ã	d3	Ó
84	ind	94	cch	a4	¤	b4	´	c4	Ä	d4	Ô
85	nel	95	mw	a5	¥	b5	µ	c5	Å	d5	Õ
86	ssa	96	spa	a6		b6	¶	c6	Æ	d6	Ö
87	esa	97	epa	a7	§	b7	·	c7	Ç	d7	×
88	hts	98	␣	a8	¨	b8	ˆ	c8	È	d8	Ø
89	htj	99	␣	a9	©	b9	˙	c9	É	d9	Ù
8a	vts	9a	␣	aa	ª	ba	°	ca	Ê	da	Ú
8b	pld	9b	csi	ab	«	bb	»	cb	Ë	db	Û
8c	plu	9c	st	ac	¬	bc	¼	cc	Ì	dc	Ü
8d	ri	9d	osc	ad	shy	bd	½	cd	Í	dd	Ý
8e	ss2	9e	pm	ae	®	be	¾	ce	Î	de	Þ
8f	ss3	9f	apc	af	¯	bf	¿	cf	Ï	df	ß

Abbreviation legend

The abbreviations in the character set tables specify the names of special characters in the character set. These codes are described in ISO 6429-1983(E), and in Annex B which describes the relationship between ISO 8859 and ISO 6937/2. They are:

Character	Meaning
ind	index
nel	next line
ssa	start of selected area
esa	end of selected area
hts	horizontal tab set
htj	horizontal tab with justification
vtb	vertical tab set
pld	partial line down
plu	partial line up
ri	reverse index
ss2	single shift 2
ss3	single shift 3
dcs	device control string
pu1	private use 1
pu2	private use 2
sts	set transmit state
cch	cancel character
mw	message waiting
spa	start of guarded protected area
epa	end of guarded protected area
csi	control sequence indicator
osc	operating system command
pm	privacy message
apc	application program command
nbspc	no-break space
shy	soft hyphen

ldfcn.h header file

The `<ldfcn.h>` header file specifies constants, macros and structures used by routines that access COFF (*Common Object File Format*) (q.v.) files.

See also “Header file” and “filehdr.h header file” in this *Encyclopedia* and the manual pages for `ldopen(S)` and `ldfcn(FP)`.

lex

lex(CP) is a “lexical analyzer generator”. It generates a portion of a C program that implements a lexical analyzer, based on specifications provided by the user. Because the specifications are relatively terse and easy to write, **lex** dramatically reduces the amount of time required to produce the analyzer code.

The core of a **lex** specification is a set of *regular expressions* (q.v.) that describe the classes of tokens that are to be found in the lexical analyzer’s input stream. These expressions are usually accompanied by a fragment of C code that describes an action to be executed when a token is recognized. The primary function of the lexical analyzer is often to return tokens to its calling routine, which is typically a parser.

For more information on **lex**, refer to the **lex**(CP) manual page or the chapter on **lex** in the *Programming Tools Guide*. See also “lexical analysis” and “parsing” in this *Encyclopedia*.

Lexical analysis

Lexical analysis involves reading a *stream* (q.v.) of characters and recognizing the elementary symbols, called tokens. Any lexical analysis (whether done by a person or a computer) is based on a set of rules that describe the token classes. For example, a string of characters composed entirely of digits might be regarded as a “number” token. *Regular expressions* (q.v.) provide a powerful notation for describing token classes. In addition to separating the input into tokens, routines that perform lexical analysis often do other processing such as associating a value with an identifier or printing a message if a particular token is recognized.

The input to a lexical analyzer is often *source code* (q.v.) written in a programming language. In this case, the tokens are the basic syntactic elements of the language. Note, however, that lexical analysis does not involve an analysis of the syntactic validity of the input stream. That process is called *parsing* (q.v.). The token classes of a programming language might include the reserved words of the language, identifiers, constants, literals, operators, and so forth.

The *lex* (q.v.) command generates C code for a lexical analyzer, based on a set of specifications provided by the user.

Library routines

Library routines are an aspect of the Development System that complement the *system calls* (q.v.). In some cases, the library routine actually makes one or more system calls. For example, the **fopen(S)** library routine calls the **open(S)** system call; the **sleep(S)** library routine calls the **alarm(S)**, **pause(S)**, and **signal(S)** system calls. The functionality of library routines and system calls may appear to be similar, and in fact, a system call is actually a form of library routine. The difference is that system calls are basic operating system functions—they provide direct access to the kernel. Library routines are built on top of system calls and often provide additional useful functionality, checks, and clean-up operations to the programmer. As a general rule, the library routines execute more slowly than the corresponding system calls because of the added functionality that they provide.

System libraries

The Development System includes several libraries of routines, with several versions of each library to support the various environments that are available. The following table lists the versions of the C library for each environment. The library selected is determined by options supplied to the **cc(CP)** command line when the program is compiled.

C System libraries for various environments

UNIX	ANSI	POSIX	XPG
libc*	libcA	libcP	libcX

- * A shared version of this library is provided in the */shlib/libc_s* library, specified with the **-lc_s** option to the **cc(CP)** command.

The following restrictions apply to using library routines:

- Never define symbols in your *application* (q.v.) that have the same names as symbols in a library, for example, **printf(S)** and **strcmp(S)**. Replacements that are incompatibly defined can cause any library, shared or unshared, to behave incorrectly.
- Do not use undocumented library routines. Use only the functions and data mentioned on the manual pages describing the routines in section (S) of the *Programmer's Reference Manual*.
- Do not add routines to an existing system library. A new software release will overwrite the contents of the library and your routines will be lost. Instead, create a new library for routines that are used in your application.

Common libraries

The following lists libraries commonly used with C language programs:

Library	Purpose
lib300.a	terminal-specific plot routines
lib300s.a	terminal-specific plot routines
lib4014.a	terminal-specific plot routines
lib450.a	terminal-specific plot routines
libaudit.a	C2 security routines
libc.a	standard C routines
libcA.a	ANSI C routines
libcP.a	POSIX C routines
libc_s.a	shared C routines
libcurses.a	curses(S) routines
libcX.a	XPG3 C routines
libdbm.a	database management
libevent.a	event driver routines (ev_*)
libform.a	forms(S) routines
libgen.a	error detection envelopes for system calls
libh.a	DOS cross development routines
libintl.a	internationalization routines
libisam.a	ISAM routines
libl.a	lex routines
libld.a	object module support routines
libm.a	math routines
libmalloc.a	malloc(S) routines
libmenu.a	menu(S) routines
libnsl.a	TLI routines
libnsl_s.a	TLI shared library routines
libos.a	OS routines for use in the Development System
libpanel.a	panel(S) routines
libplot.a	plot routines
libprot.a	C2 security routines
libPW.a	security routines
librpc.a	remote procedure call routines
libsocket.a	sockets routines
libtam.a	tam(S) routines
libtermcap.a	termcap(S) routines
libtinfo.a	terminfo(S) routines
libwindows.a	libwindows(S) routines
liby.a	yacc routines
libx.a	XENIX compatibility routines
libxti.a	XTI routines

See also “Shared libraries” and “Archive library” in this *Encyclopedia*.

limits.h header file

The `<limits.h>` header file specifies constants used by the system for data types and system limits.

See also “Header file” in this *Encyclopedia*.

Line discipline

A Line discipline is a set of routines that operates between high level *system calls* (q.v.) (such as `write(S)` and `read(S)`) and the hardware *device drivers* (q.v.). It provides configurable semantics associated with raw characters at a timed I/O between user *programs* (q.v.) and the hardware. Some of the facilities provided are line at a time input, special character handling, and software flow control. The special characters can be anything required by your device driver. The interaction between the routines is handled by a series of five *buffers* (q.v.) or queues that are used to store data so that another routine can access it.

The design of a line discipline is based on handling data from a terminal device in which text characters are combined with control characters, and other characters available on the keyboard. All behavior settable by the `stty(C)` command are implemented by the line discipline.

When line disciplines are included with a device driver, you can select among the installed line discipline packages using the `ioctl` (q.v.) system call. Behavior of each line discipline package can be configured with additional `ioctl(S)` calls.

Line disciplines can provide the following capabilities:

- keyboard input functions with character processing, which includes editing, signals, and flow control
- output display including character translations, expansions, and delays
- buffering between task time requests and *interrupt* (q.v.) routines
- disparity of device responses from user *process* (q.v.) requests
- ability to coordinate driver timing and process control
- implementation of a controlling terminal for process group control
- posting of keyboard signals
- a standard programmable interface for user process configuration through `ioctl`'s
- support for non-English character sets

The line disciplines provided by your system are referred to by numbers starting with zero, with a maximum of up to 255. Three line disciplines are provided in the *kernel* (q.v.). Line discipline zero is for terminal (tty) devices. Line discipline one is for *shl(C) shell* (q.v.) layers. Line discipline two is used by the mouse driver. There may also be a line discipline three added on the SCO UNIX System V/386 Release 3.2 Development System Version 4.0.

Line discipline routines

The following line discipline routines are *linesw* members. *tp* is an instance of the *tty structure* (q.v.) associated with a device.

Line discipline routines

Routine	Purpose
<i>l_close(tp)</i>	closes a device; called by <i>xxclose</i>
<i>l_input(tp,code)</i>	gets data from a device; called by <i>xxintr</i>
<i>l_ioctl(tp,cmd,arg,mode)</i>	device I/O control command; called by the driver <i>xxioctl</i> routine which calls the <i>ttiocom(K)</i> routine which then calls the <i>l_ioctl</i> routine
<i>l_mdminit(tp)</i>	services an interrupt generated by a modem; called by <i>xxintr</i>
<i>l_open(tp)</i>	opens a device; called by <i>xxopen</i>
<i>l_output(tp)</i>	outputs data to a device; called by <i>xxintr</i>
<i>l_read(tp)</i>	conveys processed data to user process; called by <i>xxread</i>
<i>l_write(tp)</i>	conveys data from user process to the output queue; called by <i>xxwrite</i>

See the *Device Driver Writer's Guide* for information on writing a line discipline.

linenum.h file header

The *<linenum.h>* header file specifies constants and structures used by symbolic debuggers to determine source code line numbers.

See also "Header file" and "Debugging" in this *Encyclopedia*.

Link editor

A link editor, sometimes called a loader, links object files to create an output object file. The input to the link editor normally consists of relocatable object files and *archive libraries* (q.v.). Less frequently, *executable files* (q.v.) are used as link editor input. The output file can be an executable or relocatable object file. An executable file produced by the link editor is sometimes referred to as an *a.out file* (q.v.). If an object file contains any undefined symbols, the link editor tries to find definitions for these symbols in the libraries or other object files that are being linked; this is called "resolving undefined symbols." If the output file is to be executable, all symbols must be resolved.

Link editing is done automatically when a compilation system such as `cc`(CP) is used to process source or object files. The link editor `ld`(CP) can also be called directly to link object files. In this situation, the user must make sure that the files are linked with the required libraries and other support routines. `ld` supports a command language that allows users to manipulate *sections* (q.v.) in object files. Refer to the “Link Editor” chapter in the *Programming Tools Guide* for information about this command language.

Link kit

The link kit is the collection of tools, object modules, and source files used by a user to rebuild a UNIX *kernel* (q.v.). Rebuilding the kernel is necessary when configuration changes are made, for example, adding a new device driver.

The link kit is stored as a file hierarchy rooted at `/etc/conf`. It contains kernel and *device driver* (q.v.) object modules and related files, and the programs used when the operating system is built. The hierarchy consists of the following directories:

<i>bin</i>	Tools used by the <code>link_unix</code> (ADM) utility to build a new version of the operating system.
<i>cf.d</i>	Configuration files, such as <code>configure</code> (ADM), the <i>mdevice file</i> (q.v.), and the <i>sdevice file</i> (q.v.).
<i>pack.d</i>	Kernel object modules; this includes modules that implement the basic system components such as the file system and the process control system, as well as <i>device drivers</i> (q.v.) for specific hardware devices. A subdirectory is provided for each component using the prefix name as the subdirectory name. Each subdirectory contains one or more of the following files: • <i>Driver.o</i> — the object module itself. Many modules are named <i>Driver.o</i> even if they are not device drivers; the directory name determines the specific module. • <i>space.c</i> — a file containing configuration dependent <i>structures</i> (q.v.) that are recompiled each time the operating system is built. • <i>tunable.h</i> — a file containing variables that can be changed for the device driver. • <i>stubs.c</i> — null definitions (“stubs”) for routines corresponding to features of UNIX that are not installed in the kernel.
<i>node.d</i>	Descriptions of <i>device special files</i> (q.v.) that are added to the <code>/dev</code> directory for a device driver.
<i>init.d</i>	Entries that are added to <code>/etc/inittab</code>
<i>sdevice.d</i>	A file for each driver that is being installed when the operating system is built. The files in this directory are used to construct the entries in the <code>/etc/conf/cf.d/sdevice</code> file.

lint

The **lint** program examines *C programming language* (q.v.) source programs to detect constructs that are likely to indicate errors. It enforces the type rules of C language more strictly than does the C compiler. You can use **lint** to enforce portability restrictions involved in moving programs between different machines or operating systems, or to detect legal C constructs that are error prone. The **lint** command accepts multiple input files and library specifications and checks them for consistency.

The **lint** library files are processed almost exactly like ordinary source files. The only difference is that functions that are defined in a library file but not used in a source file do not result in error messages. The **lint** command does not simulate a full library-search algorithm; it prints a message if the source files contain a redefinition of a library routine.

For more information on **lint** command options, see the **lint(CP)** manual page.

listen.h header file

The `<listen.h>` header file contains information used by the Network Listener Process Service (NLPS), which is a feature of the *TLI (Transport Layer Interface)* (q.v.) networking protocol manager. `<listen.h>` contains the following:

- format of the NLPS protocol message sent by client machines to a listener process when requesting a service
- **#defines** for compatibility with TLI routines, and the maximum connections per *STREAM* (q.v.)
- the **callsave** and **call_list** structures used by TLI library routines for handling multiple connection requests on a single *STREAM*.
- The **EMPTY** macro that clears the **call_list** structure.

Locale

A locale describes the particular cultural conventions, character set, and language of a specific area in which the user is situated. The UNIX operating system is used globally, so it is increasingly important that these territory dependent issues are resolved so that the working environment and program output is familiar and consistent with local customs where the program is running. The worldwide marketplace for UNIX systems presupposes that such issues as *internationalization* (q.v.), writing software that is not specific to any particular locale, and *localization* (q.v.), configuring this generic software to a specific region, are being used by system and application programmers. Some of the specifics of a particular locale that can be configured on a UNIX operating system are the character set in use, the method by which the elements in the set are sorted and collated, the method elements relate to each

other, the actual language being used, the numeric display format, monetary symbols, and *time* (q.v.) and *date* (q.v.) formats.

These locale related issues can be controlled by certain *environment* (q.v.) variables that can be set to particular languages, territory and/or codesets. For each locale defined, there exists a set of data tables associated with six different categories which address locale-specific functions and utilities. The following table provides details. The data tables themselves are stored in a hierarchical manner in */usr/lib/lang*.

Category name	Table name	Contents
LC_COLLATE	collate	chosen sorting sequence
LC_CTYPE	ctype	character attributes
LC_NUMERIC	numeric	numeric display format
LC_MESSAGES	message	yes and no responses
LC_MONETARY	currency	currency symbol used and its position
LC_TIME	time	time and date format

A different locale can be chosen for each category or, alternatively, by using the special category called LC_ALL or the *LANG environment variable* (q.v.), the default locale of all the six categories can be set. See the *setlocale(S)* manual page, the *locale(M)* manual page, or “LANG” in this *Encyclopedia* for more information about these environment variables.

Creating locale tables

Locale data tables can be created by the system administrator using the table generation routines that SCO provides as an extension to the AT&T System V UNIX operating system (see the *chrtbl(M)*, *coltbl(M)*, *mestbl(M)*, *montbl(M)*, *numtbl(M)*, and the *timtbl(M)* manual pages for further details on these routines).

Once generated, care should be taken that these tables are placed in the correct location so they can be accessed later. Locale data tables are arranged in a hierarchy referenced by the language used, geographical location, and the character set. The data table for a particular combination of a *setlocale(S)* category and locale string is located in

/usr/lib/lang/language/territory/codeset/category.

The argument *category* is the category name, in lowercase, with the initial LC stripped off. These tables in turn reference the data files that contain the compiled information about the locale. These data files are read into appropriate data structures and used in programs.

Here is an example subdirectory of `/usr/lib/lang` for the English language:

english			
uk		us	
ascii	8859	ascii	8859
ctype	ctype	ctype	ctype
numeric	numeric	numeric	numeric
time	time	time	time
monetary	monetary	monetary	monetary
messages	messages	messages	messages
collate	collate	collate	collate

The locale hierarchy is discussed in greater detail in the international documentation for the SCO UNIX System V/386 Operating System.

Accessing and changing the locale variables

All of the category-specific locale environment variables can be accessed or changed at the shell level or from within a program itself. To access or change an environment variable from within a piece of code, the `setlocale(S)` routine should be used. See the `environ(M)` manual page for details on how to change the environment. The `locale(M)` manual page provides details for the syntax and possible values for the values of these environment variables.

There are two other methods that locale specific information can be accessed and viewed but not altered. The first way is to access the `langinfo` database using the `nl_langinfo(S)` routine. The `langinfo` database includes the current locale setting's day names and abbreviations, month name and abbreviations, numeric formatting information, monetary value formatting information, and date/time formatting information. See the `nl_langinfo(S)`, `langinfo(F)`, and the `nl_types(F)` manual pages for more information on accessing the `langinfo` database.

The other way to access locale related information is by using the `localeconv(S)` routine to access the `lconv` structure. This structure contains locale-specific details about numeric quantities, monetary quantities, and message formats. See the `localeconv(S)` and the `lconv(FP)` manual pages for further details on how to use the `localeconv(S)` routine.

Setting your locale

There are three different methods for setting the locale variables to a particular regional requirement. When a particular locale variable is needed, its value will be extracted from the first non-NULL entry in the following list (thus, the top entry assumes precedence over the others and the bottom entry is used only if the top two are NULL):

- LC_ALL is defined in the environment and is not NULL
- the particular locale variable desired is defined in the environment and is not NULL
- the LANG environment variable is defined in the environment and is not NULL

If the resulting value is a supported locale, its information is extracted from the pre-defined database and used. If the resulting value is not supported, the value may be used for an implementation specific interpretation only.

locale.h header file

The *<locale.h>* header file specifies constants, structures and routines used for *internationalization* (q.v.).

See also “Header file” in this *Encyclopedia* and the manual page for *lconv*(FP).

Localization

Localization is concerned with configuring software to the conventions of a particular culture or *locale* (q.v.). These conventions include the region’s language, as well as the character set, monetary unit, numbering schemes, and date and time formats.

Localization can be achieved in one of two different ways, depending upon how the software was originally designed. If the software was designed and written for a particular locale, the source code will have to be changed and recompiled for use in another locale. If, on the other hand, the software was designed and written with internationalization and localization in mind, and was written without any assumptions about the locale, it will not need to be changed or recompiled. In the latter case, the code is said to be *internationalized* (q.v.) and it is designed to read pre-compiled libraries, pre-defined datafiles, and/or pre-set *environment* (q.v.) variables. If this external information is provided, the code can be left unaltered and all of the locale-specific concerns will be handled at run-time.

It is preferable for code to be localized using the latter method because it involves much less work and maintenance. The developer need only maintain these outside data storage elements as opposed to recoding and recompiling software. Source code changes are often time consuming and sometimes completely impractical.

Some of the more commonly used methods for storing locale-specific data outside of application programs are *message catalogs* (q.v.), the `LC_*` environment variables (see the `setlocale(S)` manual page), and other related data tables and environment variables. For an example of how to construct and use message catalogs within code see *Developer's Topics*.

lock.h header file

The `<sys/lock.h>` header file specifies constants used for locking processes and executable programs.

See also “Header file” and “Process control routines” in this *Encyclopedia*.

Locking files and records

See “File and record locking”

M

m4

The primary function of the **m4** macro processor is to replace one string of text with another, similar to what the C *preprocessor* (q.v.) does with the **#define** statement in C. **m4**, however, is more powerful than the C preprocessor. **m4** provides macros with arguments, conditional macro expansions, arithmetic expressions, file-manipulation facilities, and string-processing functions.

Refer to the *Programming Tools Guide* for information about invoking **m4**, defining macros, quoting, using arguments, and so on.

Macro

In general, a macro is a function which is interpreted symbolically, that is, the function call is replaced by a sequence of characters. This replacement is called "macro substitution." Some programs that use macros are **make**(CP), **m4**(CP) (a general-purpose macro processor), and the C compilation systems **cc**(CP) and **gcc**(CP). The remainder of this article will describe macros in C.

C macros

A C macro is an identifier defined with the C preprocessor **#define** directive. Like a *function* (q.v.), a macro can be defined to take arguments: when the macro substitution takes place these replace the formal parameters in the macro definition.

Note the following important differences between macros and functions:

- A macro is not executable code; the only thing a macro does is get replaced by text.
- Macros execute faster than a corresponding function. Macros are expanded (replaced by their definitions) during preprocessing, so the overhead required for a function call is eliminated.
- Macros can increase the size of the program, particularly when they appear several times, because they are expanded each time they are called. By contrast, a function is defined only once, regardless of the number of times it is called.

- Macro arguments with side effects may cause unexpected results; this happens if the macro is defined so that arguments are evaluated more than once.

For example, suppose a macro is defined as follows:

```
#define SQUARE(x) (x)*(x)
```

The result of the macro call in the following example will be 30, not 25:

```
i=5;
j=SQUARE(i++);
```

This occurs because when the macro is substituted, the macro call will look like this:

```
(i++)*(i++)
```

The expression “i++” is evaluated during the execution of this code, leading to an unexpected result.

- A macro identifier does not have the same properties as a function identifier. In particular, a macro identifier does not evaluate to an address as a function identifier does. It may be the case that after macro substitution takes place, the resulting expression evaluates to an address.
- Type checking cannot be performed on macro arguments (see the next section for how arguments should be coded). The compiler does, however, detect cases where the wrong number of arguments are specified for a macro.

Macro coding

A macro expansion involves literal replacement of arguments. As a result, extreme care must be taken to guard against misinterpretation. For example, each use of an argument should be surrounded by parentheses to ensure that macro expansion occurs correctly. An incorrectly coded macro follows:

```
#define max(a,b) (a<b ? b : a)
```

Suppose that the routine is called as **max(A&B, C&D)**; the test in the incorrect macro expands as:

```
A&B < C&D
```

Unfortunately, the less-than operator has a higher precedence than “&” and the test in the expanded macro is interpreted as:

```
(A& (B < D) &D)
```

A better way to code the macro is:

```
#define max(a,b) ((a)<(b) ? (b) : (a))
```

Magic number

Magic numbers are used throughout UNIX operating systems to indicate the type of a certain entity. These numbers are called magic because the value is arbitrary. For example, one magic number indicates the type of an executable file, and another defines the architecture used for a particular filesystem.

For a list of magic numbers used on the system, see the */etc/magic* file.

Major and minor numbers

The major and minor numbers are the high-order and low-order bytes of the *device number* (q.v.). These numbers are used in a *device driver* (q.v.) to identify a device. The major number identifies a type ("class") of device and the minor number is interpreted by the driver. Often it identifies a particular physical device (such as a unit or a controller).

The major number is a number between 0 and 255. In most cases, the system dynamically assigns a major number to each driver at boot time. It is possible to set the major number in the the major number can be set in the *mdevice* (q.v.) file when the device driver is created, but this is rarely done.

The minor number contains device-specific information assigned by the driver programmer on a bit-by-bit basis. Each bit in the eight-bit minor number can be given a different meaning depending on the needs of the device driver. User-level programs can use the *stat(S)* system call to determine the minor number of a device. The device driver kernel routine, *minor(K)* is used in a device driver to obtain the minor number from the device number.

The major and minor numbers can be examined from the command line with the *ls -l* command on the *device special file* (q.v.). For example, the following is a listing for a character driver:

```
crw-rw-rw-  4 bin      bin      52,  2 Feb 22  1989 cga
```

The fifth column **52** is the major number, and the **2** that follows is the minor number.

From a C program, the major number for a device can be obtained with the *uadmin(S)* call and the *A_GETDEV* option.

In a device driver, the major number is obtained using the *major(K)* routine, which separates the major number from the device number. In the operating system, the major number is used to index the *bdevsw* and *cdevsw* tables, which provide internal system access to block and character devices, respectively.

make

make(CP) is a general-purpose file-maintenance utility. It analyzes dependency relations between files that are processed to create a target. **make** determines whether target files have become out-of-date with respect to their sources. (A target is out-of-date if one or more of its sources has been modified more recently than the target.) **make** generates commands, which are then executed, to re-process these files and re-create the target.

For example, an executable program (the final target) depends on the object modules (.o files) that comprise it. The object modules depend on associated source files. After a source file has been changed, **make** can determine how to recreate the object modules that depend on it, and then how to recreate the final target.

The advantages of using **make** to maintain a system are that it identifies out-of-date files and allows you to re-process only those files that are needed to keep the target up-to-date.

Lists of dependencies and rules for processing source files to create targets are put in a file called a "makefile". When **make** is invoked, it reads the makefile, identifies files that need to be processed, and uses the specified rules to direct the processing.

make has knowledge of filename suffix conventions, which assists it in mapping source files to target files. For example, if an object file (with extension .o) is out-of-date, **make** knows that the object file should be regenerated by compiling (with the C compiler) a file which has the same name but a .c extension. Other filename conventions signal **make** to call processors such as *lex* (q.v.), *yacc* (q.v.), and *masm*. These conventions can be modified and extended by the user. Additionally, **make** can be directed to extract files from an SCCS (q.v.) file or library archive.

For more information on the features of **make**, see the "Make" chapter in the *Programming Tools Guide*, **make**(CP), and *Managing Projects with Make* from O'Reilly & Associates, Inc.

malloc

The Development System distributes two versions of **malloc**. The file */lib/libc.a* contains the standard **malloc**. The file */lib/libmalloc.a* contains the alternate **malloc**. The **malloc**(S) manual page documents both.

Running a program that uses many **malloc** and **free** calls on the UNIX System V/386 Operating System can cause an excessive *page* (q.v.) *swapping* (q.v.) problem because **malloc** must search the entire list of allocated and free blocks. If the program uses **malloc** and **free** calls extensively, we suggest you consider using the alternate **malloc** found in */lib/libmalloc.a*.

The alternate **malloc** maintains a separate list of free blocks and can run faster; however, data in any allocated block is immediately overwritten when the block is declared free. The standard **malloc** preserves data between consecutive **free** and **malloc** calls. Some programs rely on this functionality.

malloc.h header file

The `<malloc.h>` header file specifies constants, type definitions, structures and routines used to allocate main memory.

See also “Header files” in this *Encyclopedia* and the manual pages for **malloc(S)** and *mallinfo(FP)*.

Manual pages

The manual pages are the traditional repository for information about UNIX operating systems. Manual pages can be accessed online with the **man(C)** command (or with the **xman(X)** command on SCO Open Desktop systems) and are also printed in the *User's Reference* volumes.

In the early days of UNIX operating systems, all system manual pages were bound in one volume. As the contents of the operating system grew, the number of pages also grew. Today, it is necessary to use several volumes to hold all the system manual pages. In addition, you might find manual pages included in the documentation for other software packages installed on your system, and you might want to write your own manual pages to document commands and files that have been developed specifically for your organization.

Reference manual sections

The manual pages are arranged in sections that identify whether the term is a command, a routine used in a program, a file or structure, and so forth. For example:

Users can change access permissions for a file with the **chmod(C)** command. Programs can also be coded to modify these permissions by using the **chmod(S)** system call.

chmod is both a command and a system call; the section name given in the parentheses provides a quick identifier as to which is being discussed.

In the manual pages, the section identifier is enclosed in parentheses after the title of the manual page; this same system is used in text when discussing manual pages.

The manual pages sections are defined as follows:

User's Reference

- (C) user commands that can be executed by all users
- (M) miscellaneous topics
- (F) file formats

Administrator's Reference

- (ADM) administrative commands. Most of these require super user privileges.
- (HW) hardware dependencies, mostly special device files and peripheral devices accessed through special device files

Programmer's Reference Manual (Development System)

- (CP) programming commands that are part of the Development System
- (XNX) Development System commands for use only when doing XENIX system cross-development
- (S) system calls, library routines, and macros that are used to code user-level code
- (FP) file formats used with CP commands and Section S services
- (DOS) Routines used for MS-DOS and OS/2 cross development (available only in printed form, not online)

Device Driver Writer's Guide

- (K) kernel functions used in device drivers and other kernel-level code

Other system vendors used arabic numerals to identify the sections. SCO has replaced this convention with mnemonic character section names. However, when using commercial documents, you may come upon these other vendors' conventions. The following table correlates the numeric section numbers to the lettered section names.

Numeric sections and lettered section comparison

Section number	Section name	Contents
(1)	(C)	Commands
(1P)	(CP)	Commands sold as part of the Development System
(1M)	(ADM)	Commands requiring super user privileges
(2)	(S)	System calls
(3C)	(S)	Library routines in the C library
(3M)	(S)	Library routines in the math library
(3N)	(S)	Library routines in the Transport Layer Interface library
(3S)	(S)	Library routines in the standard I/O library
(3X)	(S)	Miscellaneous library routines
(D3X)	(K)	Kernel functions used in device drivers
(4)	(F)	System files and data structures
	(FP)	Data structures used with Development System facilities
(5)	(M)	Miscellaneous information
(6)	—	Games
(7)	(HW)	Special device files

Permuted index

Within each section, pages are arranged alphabetically. However, sometimes one page contains two or more related commands, calls, and so forth but is alphabetized under only one. The manual pages are printed with a permuted index that helps you determine the page containing the command, call, or file that you need. and can help you identify the page you need.

The dictionary definition of a permuted index is “one that is arranged in all possible ways.” Every manual page has a NAME field that names all entities described on the page and gives short descriptions of their functionality. The permuted index works with the information in that field, and lists it in alphabetical order by every substantive word given. For example, the NAME field of the `stat(S)` manual page reads:

`stat, fstat — gets file status`

A permuted index for this NAME field is shown below. Note that it is alphabetized by the second column:

<code>statfs(S), fstatfs(S)</code>	<code>gets file status.....</code>	<code>stat(S)</code>
<code>stat(S)</code>	<code>fstat(S) gets file status.....</code>	<code>stat(S)</code>
	<code>stat(S), fstat(S)</code>	<code>stat(S)</code>
<code>stat(S) fstat(S)</code>	<code>gets file status</code>	<code>stat(S)</code>

Most people find the permuted index a strange convention at first, but once one becomes accustomed to it, it can be an invaluable tool. For instance, to find all commands, system calls, and library routines that concern files, one looks in the “files” section of the permuted index. Similarly, by looking in the “SCCS” section, one can locate all pages that contain information about working with SCCS, the Source Code Control System.

Directory structure for online files

The online manual page files are located under the `/usr/man` directory in sub-directories with names such as `cat.CP` and `cat.S`. Files are displayed with the `man(C)` command.

Math routines

The following math routines let you perform common mathematical calculations. All math routines work with floating-point values and, therefore, require floating-point support. Function declarations for the math routines are given in the `<math.h>` include file.

Miscellaneous functions

Routine	Description
<code>abs(i)</code>	calculates absolute value of i
<code>cabs(x)</code>	calculates absolute value of complex number x
<code>ceil(x)</code>	calculates the smallest integer greater than or equal to x
<code>div(x, y)</code>	divides integer numerator by denominator, compute the quotient and the remainder
<code>erf(x)</code>	calculates error function of x
<code>erfc()</code>	calculates 1.0 minus <code>erf(x)</code> ; complementary error function
<code>fabs(x)</code>	calculates absolute value of x , $ x $
<code>floor(x)</code>	calculates largest integer less than or equal to x
<code>fmod(x, y)</code>	calculates floating point remainder of the division of x by y
<code>ldiv(x, y)</code>	divides long integer numerator by denominator, compute the quotient and the remainder
<code>matherr(x)</code>	handles math errors

NOTE The `matherr` routine is invoked by the math functions when errors occur. This routine is defined in the library, but can be redefined by the user if different error-handling procedures are desired. The user-defined `matherr` function, if given, must conform to the specifications given in `matherr(S)`. If no definition is present, the default error returns for each routine used. See `matherr(S)` in the *Programmer's Reference Manual*.

Bessel functions

Routine	Description
double j0(x) double j1(x)	returns Bessel functions of x of the first kind of orders 0 and 1, respectively
double jn(n,x)	returns Bessel functions of x of the first kind of order n
double y0(x) double y1(x)	returns Bessel functions of x of the second kind of orders 0 and 1, respectively
double yn(n,x)	returns Bessel functions of x of the second kind of order n

Exponential functions

Routine	Description
exp(x)	calculates exponential value of e
frexp(value,eptr)	returns the mantissa of a double value and stores the exponent indirectly in the location pointed to by <i>eptr</i> .
gamma(x)	calculates log gamma
ldexp(value,exp)	calculates $value * 2^{exp}$
log(x)	calculates natural logarithm of x ; x must be positive.
log10(x)	calculates base-10 logarithm of x ; x must be positive.
modf(value,iptr)	calculates the signed fractional part of <i>value</i> and store the integrals part indirectly in the location pointed to be <i>iptr</i>
pow(x,y)	calculates x^y ; if x is zero, y must be positive; if x is negative, y must be an integer.
sqrt(x)	calculates the non-negative square root of x

Trigonometric functions

Routine	Description
acos(x)	calculates arc cosine of x
asin(x)	calculates arc sine of x
atan(x)	calculates arc tangent of x
atan2(y,x)	calculates arc tangent of y/x using signs of both arguments to determine the quadrant of the return value
cos(x)	calculates cosine of x measured in radians
cosh(x)	calculates hyperbolic cosine of x
hypot(x,y)	calculates hypotenuse of right triangle; $\sqrt{x^2 + y^2}$
sin(x)	calculates sine of x measured in radians
sinh(x)	calculates hyperbolic sine of x
tan(x)	calculates tangent of x measured in radians
tanh(x)	calculates hyperbolic tangent of x

IEEE floating point environment control

Routine	Description
fp_rnd fpgetround()	returns current rounding mode
fp_rnd fpsetround(md_dir)	sets rounding mode and return previous rounding mode
fp_except fpgetmask()	returns current exception mask
fp_except fpsetmask(mask)	sets exception masks and returns previous setting
fp_except fpgetsticky()	returns current exception sticky flags
fp_except fpsetsticky(sticky)	clears the exception sticky flags and returns previous setting

See the appropriate manual page for more information on the math functions. Note that many of these functions are grouped together on a page, so you may need to use the permuted index to find the appropriate page when using the *Programmer's Reference Manual*.

math.h header file

The `<math.h>` header file specifies constants, structures and routines used by the *math routines* (q.v.).

See also “Header file” in this *Encyclopedia*.

m_defs.h header file

The `<nls/m_defs.h>` header file specifies constants and structures used for the *internationalization* (q.v.) (native language support) of message handling routines.

See also “Header file” in this *Encyclopedia*.

mdevice file

The `/etc/conf/cf.d/mdevice` file is a master feature listing of all the *device drivers* (q.v.) that are being added to the kernel by the *link kit* (q.v.). When the link kit creates a new operating system image (stored in the `/unix` file), it first goes to the *sdevice file* (q.v.) and determines what drivers are requested to be added to the new operating system image. The link kit then takes the information provided in the *mdevice* file to determine the attributes of the device driver, such as what routines it contains, characteristics of the device driver, the device number, and the *DMA (Direct Memory Access)* (q.v.) channel.

Refer to the *mdevice(F)* manual page for a complete listing of the field information.

Memory allocation routines

The following memory-allocation routines let you allocate, free, and reallocate blocks of memory. They are declared in the include file *malloc.h*.

Memory allocation routines

Routine	Description
<code>calloc</code>	allocates storage for array
<code>free</code>	frees a block allocated with <code>calloc</code> , <code>malloc</code> , or <code>realloc</code>
<code>malloc</code>	allocates a block
<code>realloc</code>	reallocates a block
<code>sbrk</code>	resets break value

The `calloc` and `malloc` routines allocate memory blocks. The `malloc` routine allocates a given number of bytes, while `calloc` allocates and initializes to 0 an array with elements of a given size.

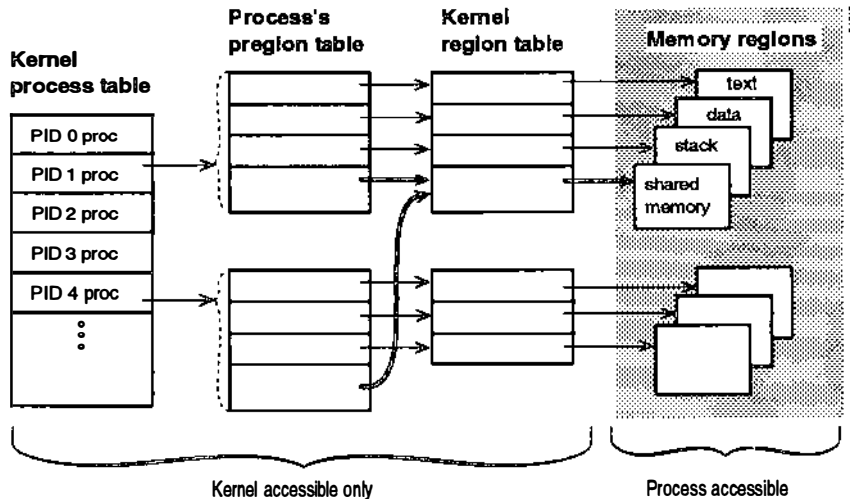
The `realloc` routine reallocates a memory block, either by changing its size or changing its location; the contents of the block remain unchanged.

A low-level memory-allocation routine is provided by `sbrk`. It increases the program's break value (the address of the first location beyond the end of the default data segment), allowing the program to take advantage of available unallocated memory.

NOTE In general, a program that uses the `sbrk` routine should not use the other memory-allocation routines, although their use is not prohibited. In particular, using `sbrk` to decrease the break value may cause unpredictable results from subsequent calls to the other memory-allocation routines.

Memory management

The memory management system manages all physical memory on the system for all executing. Each process is described to the system by some of the information in the data structures, defined in the `<sys/proc.h>`, `<sys/user.h>` and `<sys/region.h>` header files respectively. The following figure illustrates the relationship between the `proc`, `pregion`, and `region` tables:



Memory management data structures

Note that two processes are using the same shared memory segment, so both `pregion` tables point to the same member of the kernel region table for the shared memory.

The `fork(S)` and `exec(S)` system calls are intimately associated with the memory management data structures. When a process forks, the kernel:

- makes a new entry in its process table for the *child process* (q.v.), copying most of its contents from the parent's `proc` structure
- copies the parent's `pregion` table to the `pregion` table it allocates for the child
- allocates new `pregion` entries and page descriptors for the child's data and stack segments

If either the child or the parent writes to one of the pages for data or stack, the kernel makes a separate copy of that physical page for the child, referred to as "copy-on-write".

For more information about UNIX memory management, see *Design of the UNIX Operating System* by Maurice Bach.

Memory models

`cc` can create programs for five different memory models: small, medium, compact, large, and huge. In addition, small model programs generated for architectures other than 80386/80486 can be pure or impure. If an impure memory model is used then text (code) and data are allocated together, and the text is writeable; a pure model does not allow writeable text. On the 8086 and 80286 processors, these various segmentation models allow programs with text or data larger than 64K bytes. Since the 80386/80486 can address segments larger than 64K bytes, the medium, compact, large, and huge models are not required for those processors. The only memory model required (and permitted) for programs compiled for the 80386/80486 and running under UNIX is the small pure model. On the 80386 both text and data exist in the same 32-bit addressable segment; the integrity of the text portion is maintained by the operating system.

NOTE These memory models are used for XENIX, DOS, and OS/2 cross-development only.

These are the different kinds of memory models available for the non-80386 architectures:

Impure-Text Small Model

These programs occupy one 64K byte physical segment in which both text and data are combined (sometimes called the "tiny" model). `cc` creates impure small model programs by default.

Pure-Text Small Model

These programs occupy two separate 64K byte physical segments: one for text and one for data. The text is read-only and may be shared by several processes at once. The maximum program size is 128 Kbytes.

Medium Model

These programs occupy several physical segments, but only one segment contains data, while text is divided among as many segments as required. Special calls and returns are used to access functions in other segments. Data must not exceed 64K bytes. Medium model programs are always pure.

Compact Model

This model has a maximum of 64K of text, but can have any number of data segments.

Large Model

These programs occupy several physical segments with both text and data in as many segments as required. Special calls and returns are used to access functions in other segments. Special addresses are used to access data in other segments. Text and data may be any size, but no data item may be larger than 64K bytes. Large model programs are always pure.

Huge Model

These programs occupy several physical segments with both text and data in as many segments as required. It is possible to allow a data construct that spans 64K byte segments. This implementation imposes limits on the way the data construct is put together and where it is located in memory. Huge model programs are always pure.

Refer to the *Programmer's Reference Manual* for information on the `cc(CP)` options required to select the memory model.

Object files can be linked only with object and library files of the same model. You cannot combine different model object files in one executable program. `cc` automatically selects the correct model versions of the standard libraries based on the configuration option. It is up to users to make sure that all of their object files and private libraries are compiled with the appropriate model.

The special calls and returns used in medium, large, and huge model programs may affect execution time. In particular, the execution time of a program that makes heavy use of functions and function pointers, compiled with one of these models, may differ noticeably from small model programs.

In medium, large, and huge model programs, function pointers are 32 bits long. In compact, large, and huge model programs, data pointers are 32 bits long. Programs making use of such pointers must be written carefully to avoid incorrect declaration and use of these variables.

The `-NM`, `-NT`, and `-ND` options may be used with compact, medium, large, and huge model programs to direct the text and data of specific object files to named physical segments. All text having the same text segment name is placed in a single physical segment. Similarly, all data having the same data segment name is placed in a single physical segment.

memory.h header file

The `<memory.h>` header file specifies routines used for memory manipulation.

See also "Header file" and "string.h header file" in this *Encyclopedia* and the manual page for `memory(S)`.

menu.h header file

The `<menu.h>` header file specifies constants, structures and routines used to develop menu-oriented user interfaces.

See also “Header file” in this *Encyclopedia*, the manual page for `menu(S)` and the *User Interfaces Guide* manual.

Message catalog

A message catalog is used to store program messages separately from the program logic, and accessible through standard table lookup techniques. By providing message catalogs in multiple languages, a single binary can be run in different language locales without modification, providing messages in the user’s native language. Message catalogs were first proposed by the /usr/group user organization and are supported as part of the *X/Open portability guide* (XPG) (q.v.) and *POSIX* (q.v.) standards.

A message catalog is compiled from a message source file with the `gencat`(CP) command. The *application program* (q.v.) can then use the following functions to access the message catalog:

Routines to access a message catalog

Routine	Description
<code>catopen</code>	opens the specified catalog file
<code>catgets</code>	retrieves message from the catalog
<code>catclose</code>	closes catalog before exit

The *NLSPATH environment variable* (q.v.) determines the location of message catalogs. The basic operating system message catalogs are under the `/usr/lib/nls` directory. Message catalogs for applications are usually stored in a separate directory.

See the appropriate manual pages, the chapter on constructing and using message catalogs in *Developer’s Topics*, and Chapter 5 of the *X/Open Portability Guide: XSI Supplemental Definitions* for more information about message catalogs.

Messages queues (IPC)

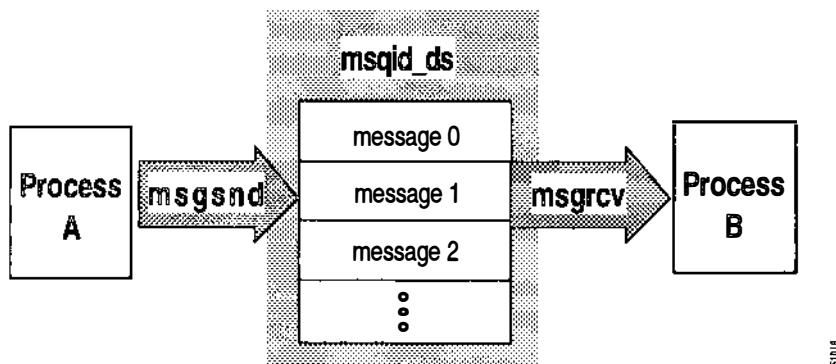
Messages queues are an *IPC facility* (q.v.) that enable cooperating processes to exchange data through buffered messages. The following table summarizes the system calls used with this facility; see the `msgget(S)`, `msgop(S)`, and `msgctl(S)` manual pages for more details.

System V message facility

Routine	Description
<code>msgget(key, msgflg)</code>	creates message queue or gets key of existing message queue
<code>msgsnd(msqid, msgp, msgsz, msgflg)</code>	sends message to queue
<code>msgrcv(msqid, msgp, msgsz, msgtyp, msgflg)</code>	receives message from queue
<code>msgctl(msqid, cmd, buf)</code>	message controls operations

Structures used for messages

The message facility uses a message queue, into which messages are sent and from which messages are received. The following figure illustrates a simple message queue used for communication between two processes.



Sending and receiving messages

The UNIX system implementation of messages dynamically allocates messages on the queue as they are needed. The `msgget(S)` routine creates the `msqid_ds` structure, and each `msgsnd(S)` call creates a new message on the queue. The `msgrcv(S)` call removes the message from the queue.

Tunable parameters for messages

The message facility, like all IPC facilities, uses a set of tunable parameters to control the capacity and behavior of message operations. When running applications that use messages, you will need to adjust these parameters to meet the particular needs of your application. If you are not running any applications that use the message facility, these tunables can be set to 1 to reclaim memory.

MSGMAP number of entries in the map that controls message queues. This map uses the same structure as the `sptmap` defined in `<sys/map.h>`; entries in the message map table say that n **MSGSSZ** segments of memory are free at x address. Each message queue that is initialized requires one entry in this map.

This number measures how fragmented you expect your map to get; it can be small if your applications send few large messages, but should be large if your applications send many small messages. To allow for holes in the map, **MSGMAP** should be set to double the value of **MSGMNI**, to ensure that **MSGMNI** rather than **MSGMAP** limits the number of message queues.

MSGMAX maximum size an individual message, in bytes

MSGMNB maximum size of a queue, in bytes. The size of a queue is the sum of all messages on that queue.

MSGMNI maximum number of `msgid_ds` message queues that can exist on the system at one time

MSGSSZ maximum number of bytes allowed for a single message

MSGSEG number of **MSGSSZ** segments of memory that are allocated for holding messages, which determines the maximum number of messages that can be posted to all message queues on the system. **MSGSEG*****MSGSSZ** determines the size of the data portion map space, which is the space allocated for message bodies. The data portion map space must not exceed 128 kilobytes (131,072 bytes).

MSGTQL maximum number of message headers on all message queues on the system. One header is required for each message sent, to define the message type and so forth, so this value determines the number of unread messages that can exist on the system at any time.

See manual pages, the “Advanced Interprocess Communications” chapter in Marc Rochkind, *Advanced UNIX Programming*, and the “Interprocess Communication” chapter in W. Richard Stevens, *UNIX Network Programming* for more information about using message queues. The “Interprocessor Communication” chapter in Maury Bach, *Design of the UNIX Operating System* contains information about the implementation of message queues.

Messages (STREAMS)

The *STREAMS* (q.v.) interface uses messages to pass data between neighboring modules on a stream. Each message contains a control part and a data part. User processes can read and write these messages with the `getmsg(S)` and `putmsg(S)` system calls, which are analogous to the `read` and `write` system calls, except that `getmsg` and `putmsg` allow an optional flag to be specified that identifies a message as high-priority. See W. Richard Stevens's *UNIX Network Programming* and Maurice Bach's *Design of the UNIX Operating System* for more information.

MMDF

MMDF is the electronic mail utility provided with the SCO UNIX System V/386 Operating System. MMDF stands for Multichannel Memorandum Distribution Facility and was created at the University of Delaware and enhanced by SCO as a value-added product. MMDF is a "back-end" utility for mail programs such as `mail(C)` and provides a mechanism for routing messages across local area and wide area communication networks with many differing protocols. MMDF contains a series of protocol modules that can be dynamically changed that permit MMDF to communicate with networks such as the Internet.

The installation and administration of the product is described in the *SCO UNIX System V/386 Operating System, System Administrator's Guide*. Further information on the routines in `libmmdf`, the MMDF library, can be found on the `mmdf(S)`, `llog(S)`, `phs(S)` and `tai(S)` manual pages.

mnttab.h header file

The `<mnttab.h>` header file specifies constants and structures used in the `/etc/mnttab` file, which records active mounts.

See also "Header file" in this *Encyclopedia* and the manual pages for `mount(ADM)` and `umount(ADM)`.

mon.h header file

The `<mon.h>` header file specifies constants, macros and structures used by the routines that prepare execution profiles of a program.

See also "Header files" in this *Encyclopedia* and the manual page for `monitor(S)`.

Motif

OSF Motif is a user interface defined by a set of on-screen objects that behave consistently in any application that uses them. The most common objects are buttons, text labels, and menus; all of which feature Motif's chiseled, 3-dimensional appearance.

SCO *Open Desktop* (q.v.) uses the Motif user interface components in many of its applications, such as ScoPaint and in its window manager, mwm, to provide a consistent look and feel. In SCO *Open Desktop* (q.v.), Motif user interface components are built with *X Window System* (q.v.) *widgets* (q.v.). In general, however, Motif is not dependent on any particular operating system or windowing system.

The SCO Open Desktop Development System provides libraries that allow application programmers to include Motif widgets in their X client applications without having to build them from scratch.

See also "Windows" in this *Encyclopedia*.

Motif window manager (mwm)

The *Motif* (q.v.) window manager (mwm) manages the resources that make up windows and creates the frame around each window. mwm was developed by the Open Software Foundation (OSF) and uses the Motif *widget* (q.v.) toolkit.

mouse.h header file

The `<mouse.h>` header file specifies constants, structures and function prototypes used to provide a "front-end" to the `<sys/event.h>` header file with specific definitions for a mouse device.

See also "Header file" in this *Encyclopedia* and the man pages for the section (S) routines that begin with the `ev_` prefix, for example, `ev_block(S)`.

Multibyte characters

Many languages contain more characters in their alphabets than can be accommodated with 8-bit characters. POSIX specifies multibyte characters for representing the value of each character across more than one byte. An analog to multibyte characters is the wide character which contains more bits than a standard character. A multibyte character consists of a sequence of eight-bit bytes, while wide characters have more than 8 bits per character. The `wchar_t` type defined in `<sys/types.h>` defines a wide character. Five section (S) routines are provided for handling multibyte and wide characters. These routines are all described on the `mblen(S)` manual page. The routines are:

Routine	Description
<code>mblen</code>	Get length of multibyte character string
<code>mbstowcs</code>	Convert multibyte string to wide string
<code>mbtowc</code>	Convert multibyte character to wide character
<code>wcstombs</code>	Convert wide string to multibyte string
<code>wctomb</code>	Convert wide character to multibyte character

These routines are defined in the `<stdlib.h>` header file which must be `#included` before any of the routines are used.

N

nan.h header file

The `<nan.h>` header file specifies constants, macros and structures used to test a floating point number to determine if it is not a number (NaN) as defined in the IEEE Floating Point Specification.

See also “Header file” in this *Encyclopedia* and the manual page for `isnan(S)`.

nlist.h header file

The `<nlist.h>` header file specifies structures used to provide information for the Intel 80386 processor.

See also “Header file” and “80x86-family processors” in this *Encyclopedia* and the manual page for `nlist(S)`.

NLSPATH environment variable

The `NLSPATH` *environment* (q.v.) variable defines the location of *message catalogs* (q.v.) and the naming conventions associated with message catalog files.

The default value of `NLSPATH` on SCO UNIX operating systems is:

```
/usr/lib/nls/msg/%l/%N.cat
```

“%l” is replaced by the language component defined for `LANG` (q.v.) the territory and codeset of `LANG` are ignored. “%N” is replaced by the name of the program passed by `catopen()`. So, if the language component of `LANG` is defined to “english”, `NLSPATH` for the `ls(C)` command is:

```
/usr/lib/nls/msg/english/ls.cat
```

The default value of `NLSPATH` is compiled into the internationalization library. Individual users may define their own `NLSPATH` in their *profile* files, although this is seldom necessary.

The default location and naming scheme for `NLSPATH` can vary from system to system; so users should familiarize themselves with how the variable is defined when moving to a different system.

See the *Developers Topics* chapter on "constructing and using message catalogs" to see how `NLSPATH` environment variable is used in conjunction with message catalogs.

nl_types.h header file

The `<nl_types.h>` header file specifies constants and variables used to provide an *internationalization* (q.v.) interface for the *libintl* library and programs that use the *libintl* library.

See also "Header files" in this *Encyclopedia*.

Null pointer

A null pointer is a C pointer that has a special value and does not point to any object. In C source code the value 0 (zero) is used to signify the null pointer; the compiler recognizes this value and generates the appropriate null pointer value internally. This value is implementation-dependent and could be anything, but in almost all C compilers, the value is actually 0. The null pointer is often represented by the *macro* (q.v.) `NULL`. The null pointer is not a valid data address and is therefore a useful marker for such things as the end of a linked list.



Object code

Object code is machine language code produced from *source code* (q.v.) by a compiler or assembler, or from other object files by a *link editor* (q.v.). An object file typically contains compiled code, and initialized and uninitialized *data symbols* (q.v.). Object files which have addresses that are relative to an origin, and need to be link-edited before they can be executed are called “relocatable object files.” An object file that is ready to run is an *executable file* (q.v.) or “absolute file.”

OMF (Object Module Format)

OMF is the binary format for object files and libraries on the XENIX operating system. The SCO UNIX System V/368 Development System fully supports the OMF format as well as the alternate COFF (Common Object File Format). Refer to the “XENIX cross development” article for information on compile options to create an OMF format object.

OMF has the following limitations:

- The **sdb**(CP), and **dbxtra**(CP), debuggers do not work with OMF binaries. The **adb**(CP), and **codeview**(CP) debuggers can be used with either OMF or COFF binaries.
- OMF binaries cannot utilize shared libraries.
- AT&T's Portable C Compiler (**rcc**) cannot generate OMF binaries.
- OMF binaries cannot be ported to machines that support only COFF binaries.

See also the “COFF (Common Object File Format)” article for comparative information about the two binary formats that are supported, and the “OMF” chapter in the *Developer's Topics* book for detailed information about the OMF format.

Open Desktop, SCO

SCO Open Desktop is The Santa Cruz Operation's graphical operating system. Based on SCO UNIX, SCO Open Desktop uses MIT's X Window System and Open Software Foundation's Motif User Interface. SCO Open Desktop includes an X server and an assortment of standard and SCO-developed X clients. SCO TCP/IP is provided as the network transport for access to remote clients and servers. The SCO NFS software allows the system to be used in a multitude of network environments.

SCO Open Desktop's main X client application is Desktop Manager, which presents a user-configurable, window- and icon-based view of the file system. Desktop Manager provides a rich, easy-to-learn alternative to command-line-based shells, including "drag and drop" file processing and user-configurable command menus. Users can modify the appearance and behavior of SCO clients by editing X resource specification files. The extent to which a user can customize features varies from client to client.

The SCO Open Desktop Development System provides libraries for developing applications that use the X library (Xlib), the X Toolkit (Xt), and the OSF/Motif intrinsics. Other libraries allow custom application development for TCP/IP.

Open Software Foundation (OSF)

The Open Software Foundation (OSF) is a consortium of computer industry vendors to which the Santa Cruz Operation belongs. OSF's mandate is to identify and certify technologies that can form the basis for the development of open systems. OSF's current products include the OSF/Motif windowing software for use with the X Windows System and an operating system that conforms with most major industry standards that currently exist.

Optimization

Optimization occurs when a compiler (or compilation system) performs additional processing in order to improve the quality of the compiled code; this increases compilation time but produces better code. When it is not attempting to optimize, the compiler uses a relatively straightforward mapping between *source code* (q.v.) and machine code. Optimization entails a more sophisticated approach to code generation that attempts to avoid inefficiencies that might otherwise have appeared in the compiled code. Sometimes these inefficiencies reflect flaws in the source code. In other cases they arise because an efficient code sequence is easily expressed in machine code but not easily expressed in the source language—if the unoptimized translation into machine code is used, then the efficient code sequence might be missed.

The primary goal of optimization is to make efficient use of machine resources. Most often this means speeding up the execution time of the program. There are sometimes trade-offs between execution time and code size: for example, some simple but “long-winded” code might actually execute faster than shorter but more complex code that does the same thing.

Various optimizations are available with the `cc(CP)` command. The following are some aspects of optimization:

Optimizing for speed

the default behavior is to optimize in favor of speed, rather than code size. Some speed optimizations result in longer code. For example, “no-op” instructions can be inserted to ensure that a function starts on a double-word boundary, as this can speed up execution. Data can be padded to align to natural boundaries. Removing some inefficiencies might reduce code size as well as increasing speed.

Optimizing for code size

the compiler can be directed not to perform optimizations that increase the code size.

Loop optimization

in general, it is advantageous for a compiler to reduce the number of instructions that appear within a loop. For example, the machine code equivalent for invariant source code (code that always has the same effect) can be positioned before the code for the loop, so that it is executed only once, instead of every time the loop is executed.

Relaxed alias checking

the compiler can be informed that the source code has no aliases (different pointers to the same address); this facilitates some forms of optimization. If the code does have aliases, and the compiler is told otherwise, incorrect optimizations might be performed.

Global register allocation

the optimizer tries to put frequently used values in registers, and to maintain them even across “basic blocks” (contiguous lines of code that have a single entry point and single exit point).

Optimization often involves code rearrangement, that is, the machine code that is produced does not directly reflect the structure of the source program. For this reason, optimization should be avoided if you intend to use a debugger, as the order in which instructions are executed in the compiled program may be different than in the source.

Option

An option is an *argument* (q.v.) to a *command* (q.v.) or *program* (q.v.) which modifies the behavior of the command or program. An option is usually specified by a single character preceded by a hyphen (-) or plus sign (+). It can also be followed, in some instances, by a single string which is an argument to the option. The exact form of an option depends on the command or program to which it is being supplied. Check the manual page for the command or program for the correct form.

Many, but not all, commands and programs use the following conventions for specifying options:

- Options that take no argument can usually be typed together as though they were one option. For example, the `ls` command can take both the `-a` option (list all files) and the `-l` option (long listing). If both options are to be specified at once, they could be typed together as one option rather than separately, as follows:

`$ ls -la`

- Options that take an argument must be followed immediately by that argument. For example, the command `lp` can take the `-d` option to print a file on a printer other than the default printer. The `-d` option takes the name of a printer as its argument, for example:

`$ lp -dprintername filename`

- Options that take an argument cannot be typed together after a single hyphen as non-argument options can but must appear as separate entities, following the command, on the command line.

Again, it is important to note that these conventions are not followed by all commands and programs.

OS/2 cross development environment

See "DOS and OS/2 cross development environment."

OSF/Motif

See "Motif."

OSI (Open Systems Interconnection)

The seven layer Open Systems Interconnect (OSI) model for computer communications was designed by the International Organization for Standardization (ISO) to create a framework for describing the issues of network architecture. By using this model, networks can be designed to enable heterogeneous systems to communicate.

Reference model

A discussion of the OSI Reference Model is presented here to place the Transport Layer Interface in perspective. The Reference Model partitions networking functions into seven layers, as depicted in the figure below.

Layer 7	application
Layer 6	presentation
Layer 5	session
Layer 4	transport
Layer 3	network
Layer 2	data link
Layer 1	physical

OSI reference model

Layer 1	The physical layer is responsible for the transmission of raw data over a communication medium.
Layer 2	The data link layer provides the exchange of data between network layer entities. It detects and corrects any errors that may occur in the physical layer transmission.
Layer 3	The network layer manages the operation of the network. In particular, it is responsible for the routing and management of data exchange between transport layer entities within the network.
Layer 4	The transport layer provides transparent data transfer services between session layer entities by relieving them from concerns with the mechanisms of data transfer.
Layer 5	The session layer provides the services needed by presentation layer entities to organize and synchronize their dialogue and manage their data exchange.
Layer 6	The presentation layer manages the representation of information in a machine-independent way. The application layer entities either communicate or reference this information in their communication.
Layer 7	The application layer serves as the interface between corresponding application processes that are exchanging information.

A basic principle of the reference model is that each layer provides services needed by the next higher layer in a way that frees the upper layer from concern about how these services are provided. In other words, each layer is a user of lower-level services and a provider of services to an upper layer. This approach simplifies the design of each particular layer.

Standards either have been or are being defined at each layer of the reference model. Each layer is defined in terms of a service and a protocol. A service interface standard at any layer frees users of the service from concern with the details of protocol implementation, or even the identity of the protocol used to provide the service.

Transport layer

The transport layer is the lowest layer in the reference model that provides the basic service of reliable, end-to-end data transfer needed by applications and higher layer protocols. In doing so, this layer hides the topology and characteristics of the underlying network from its users. More important, the transport layer defines a set of services common to layers of many contemporary protocol suites, including the International Standards Organization (ISO) protocols, the Transmission Control Protocol and Internet Protocol (TCP/IP) of the ARPANET, Xerox Network Systems (XNS), and the Systems Network Architecture (SNA).

A transport service interface enables applications and higher layer protocols to be implemented without knowledge of the underlying protocol suite. This is a principal goal of the UNIX System Transport Layer Interface. The Transport Layer Interface offers both protocol and medium independence to networking applications and higher layer protocols. because an inherent characteristic of the transport layer is that it hides details of the physical medium being used.

The *TLI (Transport Layer Interface)* (q.v.) and *XTI (X/Open Transport Interface)* (q.v.) provided with UNIX System V/386 are API's that offers services similar to those defined by the ISO Transport Service Definition (ISO 8072). As such, they are intended for those applications and protocols that require transport services. They are intended as generic interfaces to transport services. Because the Transport Layer Interface provides reliable data transfer, and because its services are common to several protocol suites, many networking applications will find these services useful.

P

Page

A page is a unit of memory management. It is a fixed number of sequential bytes of memory—this can be either virtual memory or physical memory. A page of physical memory is often referred to as a *page frame*. The *memory management* (q.v.) system manages the allocation of physical memory pages to virtual pages for all processes in such a way that the active pages in each process correspond to real memory, while the inactive virtual pages often do not have physical memory assigned to them. Virtual memory allows a program to operate in a convenient, large, and contiguous address space without regard to hardware limitations such as how much physical memory actually exists. The act of reading pages of memory to and from the disk is known as *paging* (q.v.).

The size of a physical page is usually defined by the hardware. On the Intel 80386 and 80486 platforms, a page is 4K bytes. One page frame exists for every 4K section of main memory seen by the *kernel* (q.v.). However, there are some CPUs that have different sized page frames.

Every address you use in a C program is within the boundary of some virtual page. Virtual memory must eventually be mapped to some page frame for it to be referenced.

See Maurice Bach's *The Design of the UNIX Operating System* for more information on page tables and layout of system memory. See also the *dbd* article in this *Encyclopedia*.

Page faults

A *page* (q.v.) fault is a hardware event that occurs when an attempt is made to access a virtual page that has no physical memory currently assigned to it. The page fault handler attempts to provide a physical page and to load it with the appropriate data. If it is successful, the *process* (q.v.) that received the page fault resumes at the instruction that faulted.

See the article *Paging* for a description of the paging process.

Paging

Paging is the process by which physical memory is moved to and from disk to provide virtual memory in excess of the amount of physical memory. Paging is the result of two mechanisms whose combined actions tend to keep frequently accessed pages associated with page frames, while seldom-used pages tend to migrate to the disk. The *vhand* (q.v.) daemon is responsible for page-out operations. When the total demand for physical memory by all *processes* (q.v.) on the system exceeds the amount of physical memory on the system, the *kernel* (q.v.) moves less-active *pages* (q.v.) (that are not locked in main memory) to disk from main memory.¹ The *page fault* (q.v.) handler is responsible for page-in operations. When a program tries to access memory that has been paged out, a page fault is generated signaling that the page should be paged in.

The free list contains page frames that are eligible to be reused. Page frames are added to the head of the free list when they are no longer needed, and to the tail of the free list when they might be needed again. Page frames are allocated from the head of the free list so that pages that might be needed again stay associated with page frames as long as possible. The pager moves page frames to the free list if they have not been accessed recently, and arranges to have their contents written to disk, so they eventually can be reallocated. If a page is accessed very frequently, the pager never sees it marked as not-accessed and never adds it to the free list. If a page is accessed often, it may be added to the free list, but is reclaimed by the page fault handler before it gets to the head of the list. Only the least-used pages get to the head of the free list where they are subject to reuse. The free list thus serves both as a source of available page frames and a cache of recently discarded pages that can be reclaimed quickly.

When required, the correct page can be acquired by the page fault handler in several ways:

- The page may be on the free list, that is, the page is still in memory even though its contents were paged out. In this situation, it will still contain the needed data. The page fault handler unlinks the associated page frame from the free list, marks it valid, and resumes the process. Reclaiming a page frame from the free list in this manner does not require waiting for a disk read to complete.

1. Processes can be locked in memory, meaning that they cannot be paged out. The kernel sometimes locks pages in memory, such as when doing physical I/O to a disk device. Moreover, a process can lock some of its pages in memory with the *plock*(S) system call or with the *SHM_LOCK* command to the *shmctl*(S) system call.

- If the page is paged out (in other words, the frame it formerly occupied has been allocated to another virtual page, perhaps in another process), the *kernel* (q.v.) allocates a frame from the head of the free list. The page fault handler blocks the faulting process and schedules a disk read to retrieve the page from disk. Later, after the page has been read, the kernel updates the frame address in the process's page table, marks the page valid, and unblocks the process.
- If the page is new (and hence has no data or backing store) a fresh page is allocated and filled with zeroes.

Note that frames containing kernel code and data are on neither the swap list nor the free list because no part of the kernel is ever paged out.

See Maurice Bach's *Design of the UNIX Operating System* for a more complete description of the algorithms used for paging.

panel.h header file

The `<panel.h>` header file specifies type definitions and routines used by the panel user interface.

See also "Header file" in this *Encyclopedia*, the manual page for `panel(S)` and the *User Interfaces Guide* manual.

Parent process ID (PPID)

The PPID is the *process* (q.v.) ID (PID) of the parent process. On UNIX operating systems, the scheduler, also known as `sched`, is always PID 0. `/etc/init` is always PID 1 and has a PPID of 0, because the scheduler is its parent process. User login *shells* (q.v.) have a PPID of 1 because `init` is their parent process.

To see the PPID of executing processes, see the `ps(C)` manual page for the `ps` options that will display them. If the `-f` option is used, the PPID for each process is displayed in the third column; it can be matched to the appropriate PID in the second column.

Parsing

Parsing means reading a sequence of tokens, determining its syntactic structure and deciding whether it is a syntactically correct sequence as determined by a set of rules, often called a language. For example, a C compiler determines whether a C *program* (q.v.) has any *syntax* (q.v.) errors. Parsing can be referred to as "syntactic analysis."

The programming tool *yacc* (q.v.) is used to facilitate the development of parsing routines.

passwd file

The */etc/passwd* file contains a one-line entry for every user who is authorized to log on to the system. Refer to the *passwd*(FP) manual page for a description of the password *structure* (q.v.) and some of its history.

Passwd file manipulation routines

The password file control routines provide you with low-level control of the *passwd* file. Only the *super user* (q.v.) can modify this file, but any user can read its contents. The *getprpwent*(S) routine and those routines associated with it read the *passwd* file to manipulate entries in the protected password database.² For information on a specific routine, see the appropriate pages in section S of the *Programmer's Reference Manual*.

Manipulating the passwd File

Routine	Description
endpwent	closes the password file
getpass	reads a password from <i>/dev/tty</i> , or from the standard input if <i>/dev/tty</i> cannot be opened
getpw	searches the password file for the specified user ID, and returns the matching line to the buffer
getpwent	reads the next line in the password file
getpwnam	searches the password file from the beginning for a matching name
getpwuid	searches the password file from the beginning for a matching user ID
lockpwn	locks the <i>passwd</i> file to prevent overlapping modifications
putpwent	writes a line on the stream in the same format as that of <i>/etc/passwd</i>
setpwent	rewinds the password file
unlockpwn	unlocks the password file

Pathname

A pathname, or "path" for short, designates the location of a *file* (q.v.) or *directory* (q.v.) in the *filesystem* (q.v.). It is made up of a sequence of one or more components, each of which is separated by a slash (/) character. All but the last component must be directory names. A file or directory name by itself is also a pathname. SCO UNIX System V/386 Release 3.2 Operating System Version 4.0 supports pathname components that are strings of 1 to 255 characters (other than the ASCII slash and null). On the XENIX operating system and early versions of UNIX System V/386 Operating System, file and directory names are restricted to strings of 1 to 14 characters.

2. UNIX System V/386, unlike XENIX and many other UNIX operating systems, does not store encrypted versions of user passwords in the *passwd* file itself, but in a "shadow" file called the protected password file. This feature of UNIX System V Release 3.2 provides more security, because only the system administrator can access the protected password database, whereas any user can read the */etc/passwd* file.

The `pathconf(S)` function, declared in the `<unistd.h>` header file (q.v.), can be used with the value `_PC_NAME_MAX` to determine the maximum size allowed for a component name, and with the value `_PC_PATH_MAX` to determine the maximum length of a pathname. Using this function allows you to write a *program* (q.v.) that runs both on systems that support long filenames and systems that support only filenames up to 14 characters.

A slash (/) at the beginning of a pathname indicates that the path is absolute. This means that the file or directory can be found by following the path from the top of the filesystem hierarchy, beginning at the *root* directory, on downward. A pathname that does not begin with a slash is known as a relative path which means that the path is relative to the current working directory. To display the current working directory see the `pwd(C)` manual page.

A pathname component consisting of one dot (.) refers to the current working directory. Thus, `./a` refers to the file or directory *a* in the current directory. A pathname component consisting of two dots (..) refers to the parent of the current directory, i.e. the directory one level closer to the root of the filesystem. Thus, `..` indicates the parent directory of the current working directory, while `../..` indicates the parent of the parent and so on. The path `a/b/./c`, for example, is equivalent to the path `a/c`.

The kernel keeps track of the current directory by saving its *inode* (q.v.) pointer for each process. Every time a process changes its directory, it supplies the kernel with the new path. The path to the new directory leads the kernel to the inode of the new current directory, which is then stored for that process.

Both absolute and relative pathnames can be used with all *commands* (q.v.) that take either a file name or a directory name as an *argument* (q.v.). The use of pathnames permits access to files and directories located elsewhere in the filesystem hierarchy without requiring that the current working directory be changed.

Permissions, file access

Permissions determine the degree of access to a *file* (q.v.) or *directory* (q.v.) allowed a particular user. Permissions are divided into three types and are applicable to three groups of users. The function of these permissions differ slightly depending on whether they are file permissions or directory permissions. The *sticky bit* (q.v.) is also set as a file permission. *C2 security* (q.v.) checks file permissions against a secured data base to detect changes.

File permissions

File permissions are set separately for the owner of the file, the group owner of the file, and for all others. Three permissions can be set for each category of user:

- r (read)** allows the file to be used as input to commands that will read its contents; for example, the file can be viewed with such commands as **cat**, **pg**, **more** or **view**. It also allows a file to be copied with the **cp(C)** command or printed with the **lp(C)** command.
- w (write)** allows the contents of the file to be modified. This can be done using a text editor such as **ed**, **vi** or **emacs** or can be done by a program. Unsetting write permission for a file does not prevent the file from being removed (**rm**), it only prevents the contents of the file from being modified. Protecting a file from removal is the function of directory permissions (see below).
- x (execute)** allows an *a.out* file (q.v.) or a shell script file to be executed.

Use the **chmod(C)** command or the **chmod(S)** system call to set file permissions. For discussions of owner and group, see the “User ID” and “Group ID” articles.

The current permissions for a file can be shown with the **ls -l** command, for example:

```
ls -l /bin/cp
-rwxr-x--x  3 bin      adm      8768 Jun 10 1990 /bin/cp
```

The last nine characters of the string **-rwx--x--x** indicate the permission settings for the file **/bin/cp**. Further subdividing the 9 character permission string into three groups of three characters, indicates, consecutively, the permissions for the owner (**rwx**), group (**r-x**) and all others (**--x**). In practical terms, this means that the **/bin/cp** file can be written only by the owner, can be read by the owner (“bin”) or the group (“adm”), and can be executed by the owner, group, and all others.

The rules for file access are determined by the process’s *effective user ID* (q.v.) as follows:

- The *super user* (q.v.) can access any file, no matter what permissions are set.
- When the process’s *EGID* (*Effective Group ID*) (q.v.) matches the user ID of the file, the “owner” bits of the permissions determines the file access.
- When the process’s EUID does not match the user ID of the file but the effective group ID does, the “group” bits of the permissions determines the file access.
- When neither the EUID nor EGID match the user ID or group ID, respectively, and the processes EUID is not super user, the “other” bits of the permissions determines the file access.

Directory permissions

Directory permissions are indicated in the same way as file permissions though the meaning changes slightly:

- Execute permission (x) on a directory indicates the ability to search or access the contents of the directory; in other words, to use the directory name as a component of a pathname.
- Read permission (r) on a directory means that the files in the directory can be listed with a command such as `ls`; implicitly, listing files in a directory requires “reading” the directory to get the file names.
- Write permission (w) on a directory means that entries (including files) can be created or deleted. Search permission (x) must also be set for this to be possible.
- The permissions on a directory control the ability to list, modify, or search the contents of the directory, but do not affect one's ability to read or execute files in the directory (which is controlled by the permissions on the individual files). Note that a process cannot open files in a directory with read but not execute permissions.
- To remove a directory with the `rmdir(C)` command, you must have write permission for the parent directory.
- Execute permissions should always be set for normal access to a directory.

Directory permissions can be shown with the `ls -ld` command, for example:

```
ls -ld /bin
drwxr-xr-x  2 bin      bin      2352 Aug 15 12:23 /bin
```

The *d* in first position of the permission string `drwxr-xr-x` indicates that `/bin` is a directory. The remainder of the string indicates that the owner can list (r), modify (w) and search (x) the directory whereas group members and all others can only list the contents of the directory (r) or access the contents (x) but cannot create or remove entries in the directory. For information on changing permissions see the `chmod(C)` manual page.

setuid (SUID) and setgid (SGID) permission

The `setuid` and `setgid` bits enable a program run by a user to access or modify a file in ways that would not normally be permitted according to the file access permissions. For example, consider the `passwd(C)` command which is used to change passwords. Changing a password involves modifying the `/etc/passwd` file, which contains all password information and is writable only by the owner, normally `root`. To allow an ordinary user to modify this file, the `passwd` command is marked `setuid` or `SUID`. This is indicated by a special permission bit on the object file for the `/bin/passwd` command. With this bit set for an executable file, the running program executes with the EUID of the file's owner for the duration of the program's execution; thus, when the `passwd` command is run it temporarily inherits `root`'s permissions and is able to update the `/etc/passwd` file with the user's new password. The `passwd` command itself verifies that users modify only their own passwords. `setgid` permission is used in the same manner as `setuid` permission but rather than

adopting the user ID of the file's owner, the process adopts the group ID of the file's group.

A program that is **setuid root** has particularly extensive privileges because root can access any file in the system. **setuid** programs need not be owned by root; any other valid UID can be used, and when executing, the program will run with the permissions of the owner.

When **setuid** permission is set, "s" appears in place of "x" for the execute permission of the owner. Similarly, when **setgid** permission is set, "s" appears in place of "x" for the execute permission of the group. The first line below shows a file with **setuid** permission; The second line shows a file with **setgid** permission.

```
ls -ld /bin
-rwsr-xr-x  2 mine      pubs      3257 Aug 15 12:23 /sam_setuid
-rwxr-sr-x  2 your      engr      1352 Aug 15 12:23 /sam_setgid
```

See also **chmod(C)**, **chmod(S)**, **setuid(S)**, **exec(S)**, **setuid(S)**, the "Trusted facilities," "Effective User ID," "Sticky bit," and "User ID" articles in this *Encyclopedia*, and Maurice Bach, *The Design of the UNIX Operating System*.

Perms list

The permissions list (perms list) contains an entry for each file that is installed or created as it is installed in a software distribution. The perms list is located in the */etc/perms* directory. The filename of the perms list is an abbreviated name (maximum eight alphabetic characters) of the product; the name of the perms list file must match the value of the **#prd** keyword in the perms list.

The perms list file has the following format:

- A pound character (#) and a space at the beginning of the line indicate a comment line.
- A pound character and an exclamation sign (!) or a pound character and a letter from a to z at the beginning of the line indicate a special keyword phrase. Only the keywords described in the next section are currently defined.
- Non-blank lines that do not start with a pound sign are description lines.

The **custom(ADM)** utility *parses* (q.v.) the keyword phrases and description lines to determine what is being installed or, in the case of directories or device files, created. There is one item specification per line.

Perms list example (Part 1 of 2)

```

1      # Small OS Perms List
2      #
3      #prd=smallos
4      #typ=386GT
5      #rel=3.2.4
6      #set="The Very Small OS Product"

7      uid    root    0
8      uid    bin     2
9      gid    root    0
10     gid    bin     2

```

lines 1-6 These lines introduce the product. Line 3 is the product name, line 4 specifies the computer processor code, line 5 establishes the release number, and line 6 provides space for a product description. The statements are:

#prd The product name is a lowercase string with a maximum of eight characters. The **prd** value has the same name as the perms list file.

#typ Vendor-defined machine environment string.

#rel Vendor-defined release string.

#set Used as a verbose product name for user interface display.

lines 7-10 Specify the *user ID (UID)* (q.v.) and *group ID (GID)* (q.v.) that are used for the root and bin logins while the product is being installed. In each line, the first field indicates the ID type (either gid or uid), the second field contains the user or group name for the ID, and the third field is the corresponding numeric ID.

These values are used in the package descriptions (such as those shown in lines 16-43).

Perms list example (Part 2 of 2)

```

11      #
12      #!ALL 300   Entire Product
13      #
14      PERM      f644   bin/bin    1   ./etc/perms/smallos 01

15      #
16      #!BASEPKG 256   Basic Package
17      #
18      # directory that is part of the required product
19      BASEPKG   d755   bin/bin    1   ./dev
20      # directory not part of the required product (D signifies)
21      BASEPKG   D755   bin/bin    1   ./tmp
22      # standard file
23      BASEPKG   x711   bin/bin    1   ./bin/cat           02
24      # standard file with 3 links
25      BASEPKG   x711   bin/bin    3   ./bin/cp           01
26      .          .      .          .   ./bin/ln           01
27      .          .      .          .   ./bin/mv           01
28      # special chmod bit set, see chmod(S)
29      BASEPKG   x2111  bin/backup  2   ./bin/df           01
30      .          .      .          .   ./etc/devnm       01
31      # shell script: note that read permissions are needed
32      BASEPKG   f755   bin/bin    1   ./usr/lib/mkdev/lp 02
33      # character device node
34      BASEPKG   c440   root/root   1   ./dev/auditr      21/0

35      #
36      #!DEVPKG   26    Development Package
37      #
38      # block device node
39      DEVPKG    b440   root/bin    1   ./dev/root        1/40
40      # linked device node
41      DEVPKG    c622   root/bin    3   ./dev/console     3/1
42      .          .      .          .   ./dev/syscon
43      .          .      .          .   ./dev/systty

```

line 14 This statement specifies the location of the perms list file and establishes that it is to be included in the **PERM** package. The perms list is always assigned to this package and must reside in the */etc/perms* directory. Refer to the “File Descriptions” section that follows for a complete description of the fields in this statement.

lines 16-43 The rest of this perms list is hypothetical and provides descriptions for two packages, **BASEPKG** and **DEVPKG** explained in the next section.

Package Declarations

Add-on application software is grouped into and installed as units called packages. A package is a logical group of files; a package groups files into functional units that can be installed as desired — either all packages together or individually. Multiple packages can be delivered on a single volume of media, or a package can span multiple volumes of media.

Each package must have a declaration containing the following fields: a package name, the package size in 512-byte blocks, and a string containing the package description.

The package named PERM is a reserved word. All products have a PERM package in their perms list that includes the files used when creating a product, except for the label file (which is not listed in the perms list). Before the PERM package declaration, the #! ALL statement is required to introduce the PERM package. (Refer to lines 12 - 14 of the perms list example.)

Package names must be unique and can be up to eight characters in length.

File Descriptions

A file description consists of a package specifier, a *permission* (q.v.) specification, owner and group specifications, the number of links on the file, the file name, and a volume number separated by tabs. Refer to the perms list example, lines 14 through 43 for examples of file descriptions. The file descriptions fields are:

- **Package specifier** — An arbitrary non-white-space string which is the name of a package within a distribution set. The string length is determined by the remaining number of characters on an 80-column line.
- **Permission specification** — a file type, followed by an octal `chmod(C)` permission specification. The file type can be one of the following characters:

Character	Meaning
x, X	Executable
a, A	Archive
e, E	Empty file
b, B	Block device
c, C	Character device
d, D	Directory
f, F	Text file
p, P	Named pipe
o, O	Okay. Indicates that there should be no file type checking allowing any format or contents in what would normally be the header section of an executable. For example, data files should be of type "o".

An uppercase file type indicates that the associated file is optional. When **custom** installs a package, it provides a test to ensure that all pieces of the product or package is installed. If a file is not fully installed, **custom** lists a package or product as "Partially Installed." A product is considered fully installed if:

1. The respective perms list file exists in */etc/perms*, and
2. All required files are installed. A required file is indicated by a lower-case file type. If the letter is in uppercase, **custom** does not consider the file as part of the criteria for a complete installation.

If all files are installed with uppercase file types, **custom** evaluates the product as installed only if the perms list for the product exists in */etc/perms*.

- **Owner and group specifications** — the owner and group permissions are in the third column separated by a slash: for example, *bin/bin*. The owner and group names are the same as those defined at the start of the perms list.
- **Links** — the number of links are in the fourth column. If there are links to the file, the following lines contain the linked filenames with no information other than the volume number. Linked device nodes do not have volume numbers.
- **File pathname** — the fifth column is the pathname of the file on the distribution media volume and is relative to the root directory. The pathname must be relative ("*./*"); that is, it must not begin with a slash "*/*".
- **Volume numbers or major-minor numbers** — the sixth column has two uses:
 - The number of the media volume on which a file exists. This column is blank for directories because directories are created as needed when **custom** executes. The media volume has the same value as the *vol=* directory of the label file. The maximum number of digits in the volume number is two (up to 99 volumes are possible).
 - In the case of special files, the *major and minor device numbers* (q.v.) are specified, separated by a slash. Device files are created by **custom** while it is executing. When a device file is linked to another device file, a value is not put in the sixth column of a linked device file entry. See lines 39 through 43 of the "Perms list example" for examples of device file entries.

Permuted index

See "Manual pages."

pic

pic is a pre-processor used with the text-formatter *troff* (q.v.) to format diagrams and pictures using a number of basic shapes. The user embeds drawing commands in a **troff** document. The file is processed by **pic**, which converts these commands into the equivalent **troff** codes. The resulting file is then processed by **troff**.

The objects that **pic** supports are box, circle, ellipse, arc, line, arrow, and spline. Size attributes such as width, height and radius can be specified as appropriate. It is possible to position objects or text at an absolute position on the page, or relative to other objects. Positions on polygonal objects may be specified by "compass points" — north, south, center, and so forth. The start- and end-points of other objects can be referenced. Objects may be labeled for later reference, or referred to sequentially. There is some programming support for **pic** commands: macros, passing an external file through a macro, loops, if-statements, and the ability to run *shell* (q.v.) commands.

troff and **pic** are not included in the SCO UNIX System V/386 Development System, but are available from a number of outside vendors or as public domain software for UNIX, XENIX, and DOS systems. For more information about **troff**, refer to O'Reilly and Associates *UNIX Text Processing*.

Pipes

A pipe is a mechanism that allows a process to pass information to another process. The operating system treats the pipe as a special type of file: each pipe has a corresponding inode, and programs manipulate pipes via a *file descriptor* (q.v.). However, data in a pipe is transient: once data has been read from a pipe, it cannot be read again. Pipes are most commonly used to pass information between programs, thus eliminating the need to create temporary files to pass information to other programs. A pipe also can be used as a temporary storage place for a single program, although this is not a good practice.

UNIX System V/386 supports two kinds of pipes: named pipes (FIFO's), that are created with the **mkfifo**(S) or **mknod**(S) system call and associated with the process with the **open**(S) system call; and unnamed pipes, that are created with the **pipe**(S) system call or **popen**(S) library routine.

The following tables summarize the system calls and library routines that provide pipe functionality.

System calls and library routines for unnamed pipes

Routine	Description
<code>pipe(fdptr)</code>	creates pipe and returns two file descriptors, one for reading and one for writing. <i>fdptr</i> is the integer array that will contain the two file descriptors.
<code>popen(command,type)</code>	creates pipe between the calling process and <i>command</i> . <i>type</i> specifies whether the pipe is for reading or writing.
<code>pclose(stream)</code>	closes pipe opened by <code>popen</code>

The kernel assigns an *inode* (q.v.) for the unnamed pipe when it is created, and allocates pairs of descriptors in the user area file descriptor table and system *file table* (q.v.).

System calls for named pipes

Routine	Description
<code>mkfifo(path,mode)</code>	make a FIFO special file (POSIX)
<code>mknod(path,S_IFIFO,dev)</code>	make a FIFO special file (AT&T SVID Issue 2)

A named pipe has a directory entry and an associated path name. Processes open a named pipe with the `open(S)` system call that references the path name; the *flag* argument to `open` indicates whether the pipe is being opened for reading or writing.

Processes use the `read(S)` and `write(S)` system calls (or the corresponding *standard I/O* (q.v.) library routines) to access both types of pipes once they are created and opened. An unnamed pipe exists until it is closed with the `close(S)` system call (for pipes created with `pipe`) or `pclose(S)` library routine (for pipes created with `popen`) or until the process exits. Named pipes exist as files on the file system independently of whether any processes are accessing them; use the `unlink(S)` system call or the `rm(C)` command to remove a named pipe.

Data in pipes cannot be accessed randomly. The byte offsets for pipes are maintained in the *inode* rather than in the user file descriptor, so `lseek(S)` operations cannot be used with pipes.

See the manual pages and the “Basic Interprocess Communication” chapter in Rochkind’s *Advanced UNIX Programming* book for information about using pipes in a program. See Maurice Bach, *The Design of the UNIX Operating System* for information about the algorithms and structures used for pipes.

poll.h header file

The `<sys/poll.h>` header file defines the `pollfd` structure and defines poll events and the number of poll file entries. These definitions are used by the `poll(S)` system call that is part of the *STREAMS* (q.v.) interface. The `<poll.h>` header file" is a front-end interface to the `<sys/poll.h>` header file, provided for compatibility with other software.

Portable programs

A portable *program* (q.v.) is one that can be compiled and executed on different machines or different compilers without alterations to the *source code* (q.v.). Since most programming languages leave many details to the implementor, these issues are often determined by the specifications of a particular compiler or machine being used. Code written with a specific compiler or machine in mind is almost always non-portable. Another major factor in determining the portability of a certain program is how much the code relies on its environment (the *system calls* (q.v.), *library routines* (q.v.), and the way they are organized within the system). Obviously, the more a piece of code relies on the environment, the less the likelihood that it will be portable to other systems where the environment is probably different.

Portability also affects data that needs to be used over different systems. One of the primary reasons that data is non-portable is because of the *byte order* (q.v.) within a word. Different systems organize bytes in different ways within multibyte words. A file consisting of C program data types such as structures, unions, or arrays can also be misinterpreted when accessed on a system different than the one that created it. This is because different systems pad these storage elements in different ways.

To make data portable, it is recommended that it be passed as an *ASCII* (q.v.) stream. This scheme will interpret the data uniformly on most systems; however, it still would not be portable to non-ASCII based systems.

The following list presents hints for writing portable programs.

- Use external data files to read machine specifications instead of making hard-coded assumptions.
- Avoid hard-coded pathnames, files, directories, or other system dependent parameters within the code.
- Avoid non-portable hard-coded language features or assumptions about the character set.
- Apply *internationalization* (q.v.) techniques such as *message catalogs* (q.v.) and *locale* (q.v.) environment variables to allow for world-wide usage.
- Avoid usage of non-standard I/O devices.
- Avoid assumptions about machine-related parameters such as the byte length, word length, or memory alignment.
- Avoid assumptions about how the compiler operates internally.

For more information on program portability, particularly C language program portability, see the "Writing portable programs" chapter in the *Developer's Topics*.

POSIX standard

The POSIX (Portable Operating System Interface) standard is the IEEE's specification for an operating system interface based on UNIX operating systems. It is also accepted by ISO (*International Organization for Standardization*) (q.v.).

POSIX is a family of standards. The first member of the family, POSIX.1, officially titled *Part 1: System application programming interface (API) [C Language]* defines the basic operating system; the official version was issued in September 1990. Official drafts that were issued in September 1986 and September 1988 are superseded by the 1990 final version.

Other committees were under active consideration at the time of publication, or are expected to become active in the near future. They are:

- 1003.0 overall guide to POSIX-based or related Open Systems standards and profiles
- 1003.1 language-independent service descriptions
- 1003.2 shell and utility facilities
- 1003.3 POSIX verification test methods
- 1003.4 real time facilities
- 1003.5 ADA language binding
- 1003.6 secure/trusted system considerations
- 1003.7 system administration
- 1003.8 transparent file access (related to 1003.1)
- 1003.9 FORTRAN language binding
- 1003.10 supercomputing applications
- 1003.11 transaction processing applications
- 1003.12 network interface facilities (XTI and sockets)
- 1003.13 real time application environments
- 1003.14 multiprocessing applications support
- 1003.15 batch environments
- 1003.16 C language bindings
- 1003.17 language independent bindings
- 1003.18 POSIX platform environment

Information about the status of these other committees and working drafts of their standards are available from the IEEE Computer Society.

Under SCO UNIX, POSIX-conformant programs can be compiled by specifying the **-posix** option to the **cc(CP)** command. The “Bibliography” gives a full citation for the published POSIX 1 standard. Users who are interested in writing application programs that are POSIX-conformant should see Donald Lewine’s *POSIX Programming Manual*, which is also listed in the “Bibliography.”

Preprocessor

A preprocessor is a *program* (q.v.) that processes an input file and generates input for another program, the target processor. For example, *eqn* (q.v.) and *tbl* (q.v.) are preprocessors for *troff* (q.v.), *grap* (q.v.) is a preprocessor for *pic* (q.v.), and **cpp(CP)** is a preprocessor for the C compiler. Finally, *m4* (q.v.) is a general-purpose preprocessor which can be used to define and process macros for a variety of applications.

Typically, the input to the preprocessor is written in the language understood by the target processor, but contains “preprocessor directives.” These directives constitute a small set of operators which extend the language of the target processor.

proc structure

Each executing *process* (q.v.) has its own *proc structure* (q.v.) (defined in the *sys/proc.h header file* (q.v.)). The *proc* structures for all executing processes are contained in the *kernel’s* (q.v.) process table, whose size is determined by the **NPROC** tunable kernel parameter.

The kernel’s process table is always in memory (never paged out), so the *proc* structures contain information the kernel may need while a process is paged out (such as its priority, its process group and parent process, address of the “u” page, and addresses used for sleep functionality).

Process

A process is one execution of a *program* (q.v.). It consists of instructions and data and a collection of information about the executing program, such as its allocated memory, open files, and current execution status. The system identifies each process with a unique ID number called a *PID (process ID)* (q.v.). Each user process runs in its own address space, separated from all other processes. Other processes may communicate with it through facilities such as *signals* (q.v.), *pipes* (q.v.), *shared memory* (q.v.), *semaphores* (q.v.), *messages* (q.v.), and *files* (q.v.), but, otherwise, it is isolated from all other processes.

Some restrictions on user address space are:

- User processes are restricted to size limits and the size of a file each can write.
- User processes, unless owned by root, are restricted by *permissions* (q.v.) modes and *security* (q.v.) limits.
- User processes contain a block of memory called the *user structure* (q.v.) (also called a “ublock” or “u_area”) which contains information about a process for use by the operating system. Each process has a private u_area.
- User processes are scheduled in and out of execution until either a process switch occurs or time slice elapses. Then the executing process is removed from execution and put to “sleep” until another time slice is available for it to execute.

Memory segments for processes

Each process is represented by two memory segments, called the “text” (or code) segment and the “data” segment, and a set of data structures referred to as the “user structure”. A text segment contains code and constant data and may be shared by all processes running the same program. The user structure records the information that the kernel needs to manage the process, such as register contents, priority, open files, and so forth. To maintain system integrity, a process cannot address its kernel environment directly, but can use *system calls* (q.v.) to modify it.

Creation and termination

When the operating system is first booted, it creates the **init** process, which is the parent of other processes created when users log in, eventually resulting in the execution of a shell. All commands executed from the shell are children of the shell.

Process states

A process can be in one of seven states; the output of the **ps(-fl)** command displays the current state:

- S process is blocked awaiting some event. This is the most common state.
- R process is runnable when the kernel schedules it, although it is not currently running
- Z. *zombie state* (q.v.), the process has issued the **exit** system call and no longer exists, but it leaves a record containing an exit code and some timing statistics for its parent process to collect. The zombie state is the final state of a process.
- T process has been stopped by a signal because the parent process is tracing it

- I process is newly created and is in a transition state; the process exists, but is neither blocked nor runnable. This state is the start state for all processes except process 0 (**swapper**).
- X process is blocked awaiting memory availability
- O process is executing in either kernel or user mode

Process attributes

The following table lists all process attributes. For each attribute, the table identifies whether the attribute is preserved across an **exec** operation, whether a copy is inherited by a *child process* (q.v.) that is created with a **fork** operation, and calls that can affect the attribute.



Process attributes

Attribute	Preserved across exec?	Copy inherited by child?	Affected by
text segment	N	Y*	exec
data segment	N	Y	exec, brk, sbrk, execution [†]
environment (in data segment)	optional	Y	exec, execution [†]
arguments (in data segment)	optional	Y	exec, execution [†]
real user-ID	Y	Y	setuid
real group-ID	Y	Y	setgid
effective user-ID	Y (N if setuid)	Y	setuid
effective group-ID	Y (N if setgid)	Y	setgid
process ID	Y	N	fork
process-group ID	Y	Y	setpgrp
parent-process ID	Y	N	fork, termination of parent
current directory	Y	Y	chdir
root directory	Y	Y	chroot
nice value	Y	Y	nice
trace flag	Y	Y (optional)	ptrace
alarm clock	Y	N	alarm
file mode creation mask	Y	Y	umask
file size limit	Y	Y	ulimit
defaulted signals	Y	Y	signal
ignored signals	Y	Y	signal
caught signals	N	Y	signal, receipt of signal
user CPU time	Y	N	execution [†]
system CPU time	Y	N	execution [†]
children's user CPU time	Y	N	termination of child
profiling flag	N	Y	profil
file descriptors	Y [‡]	Y	open, creat, close, dup, pipe, fcntl
file pointers	Y	shared**	open, creat, read, write, lseek
close-on-exec flags	Y (if off)	Y	fcntl
O_NDELAY flags	Y	shared**	open, fcntl
status code	—	—	termination
process lock	Y	N	plock
attached shared memory segments	N	Y	shmat, shmdt
semaphore adjustments	Y	N	semop

* may be shared

† “execution” indicates that the attribute may be changed by normal process execution, without making a system call.

‡ N if close-on-exec

** “shared” indicates that the child shares this attribute with its parent rather than inheriting a distinct copy.

Process pathname resolution

For each process, the operating system maintains a directory, in addition to the current working directory, to be considered the *root* directory for the purpose of resolving absolute pathname searches. A process's *root* directory need not be the *root* directory of the *root* filesystem. See the **chroot(ADM)** and the **chroot(S)** manual pages to change the *root* directory of the current process.

Process control routines

Process control routines are those routines that allow such things as the creation and termination of *processes* (q.v.), as well as interaction between processes. The following table lists the available process control routines. More information on each of the individual routines can be found on the corresponding manual pages.

Process control routines

Routine	Description
<code>abort()</code>	aborts a process
<code>alarm(sec)</code>	arranges for an alarm signal to be sent to the process at a later time
<code>exec(path, arglist, (char *)0)</code>	executes child process with argument list [†]
<code>execle(path, arglist, (char *)0, envp)</code>	executes child process with argument list and given environment [†]
<code>execlp(file, arglist, (char *)0)</code>	executes child process using PATH variable and argument list [†]
<code>execv(path, argv)</code>	executes child process with argument array [†]
<code>execve(path, args, envp)</code>	executes child process with argument array and given environment [†]
<code>execvp(file, argv)</code>	executes child process using PATH variable and argument array [†]
<code>exit(status)</code>	terminates process and returns <i>status</i> to system
<code>_exit(status)</code>	terminates process without flushing buffers
<code>fork()</code>	creates a new process
<code>getpgrp()</code>	gets process group ID number
<code>getpid()</code>	gets process ID number
<code>getppid()</code>	gets parent process ID number
<code>gsignal(sig)</code>	raises the signal; used with <i>ssignal</i>
<code>kill(pid, sig)</code>	sends a signal to a process or group of processes
<code>lock(flag)</code>	locks a process in main memory
<code>monitor(lpc, hpc, buf, bufsz, nfun)</code>	prepares an execution profile; used with <i>profil</i>
<code>nap(period)</code>	suspends a process for a period of time, or until a signal is received

(Continued on next page)

Process control routines

(Continued)

Routine	Description
<code>nice(incr)</code>	decreases CPU priority of a process
<code>pause()</code>	suspends a process until a specified signal is received
<code>proct(1,pid,command,arg)</code>	controls active processes or groups of processes
<code>profil(buff,bufsiz,offset,scale)</code>	creates an execution-time profile of a section of core memory
<code>ptrace(request,pid,addr,data)</code>	allows parent process to trace execution of a child process
<code>rdchk(fdes)</code>	checks to see if there is data to be read
<code>sbrk(incr)</code>	alters amount of space allocated to the data segment of the calling process
<code>setpggrp()</code>	sets group ID of a process
<code>signal(sig,func)</code>	allows a process to handle signals
<code>sigset(sig,func)</code>	allows a process to handle signals
<code>sleep(seconds)</code>	suspends execution of calling process for a period of time
<code>ssignal(sig, action)</code>	implements software signals
<code>times(buffer)</code>	gets execution times of processes and child processes
<code>ulimit(cmd,...)</code>	provides control over process limits
<code>wait(ptr)</code>	suspends the calling process until it traps a specified signal or until a child process stops

† The different forms of the `exec` call can be outlined as follows:

Routine	Uses PATH setting?	Argument-passing convention	Environment
<code>execl</code>	no	argument list	inherited from parent
<code>execle</code>	no	argument list	pointer to environment table for child process passed as last argument
<code>execlp</code>	yes	argument list	inherited from parent
<code>execv</code>	no	argv array	inherited from parent
<code>execve</code>	no	argv array	pointer to environment table for child process passed as last argument
<code>execvp</code>	yes	argv array	inherited from parent

Process group ID

Each active *process* (q.v.) is a member of a process group that is identified by a positive integer called the process group ID. This grouping permits the signaling of related processes; see `kill(S)`.

Process group leader

A *process* (q.v.) group leader is any process whose process group ID is the same as its process ID. Any process may become a group leader by calling `setpgid(S)`. A process inherits the process group ID of the process that created it; see the `fork(S)` and the `exec(S)` manual pages for more information.

Process ID (PID)

Each active *process* (q.v.) in the system is uniquely identified by a positive integer called a process ID. PID 0 is reserved for the scheduler, also known as `sched`, and PID 1 is reserved for the `/etc/init` process.

Process subsystem

The *process* (q.v.) subsystem handles process execution, management, communication, and synchronization. Processes are created when a process invokes the `fork(S)` system call. The forked process is called a *child process* (q.v.). Typically, systems permit from 15 to 30 processes for each user on the system. On a single processor system, the process subsystem manages these processes so that only a single process executes at a time. Processes are permitted to execute until, for example, a time limit expires or the process must wait for an event to occur, such as data arriving from a device. At this point the process is suspended until the scheduler permits the process to continue execution.

prof.h header file

The `<prof.h>` header file specifies macros used to provide support for finer granularity of profiling information.

See also "Header file" in this *Encyclopedia* and the manual page for `prof(CP)`.

Profilers

A profiler is a tool for analyzing a program's run-time behavior, a process known as dynamic analysis. There are two profilers available for dynamic analysis of *C programs* (q.v.) written in a UNIX operating system environment: **lprof** and **prof**.

The **prof** tool performs time-profiling; it reports how much time is spent executing various portions of a program. The **lprof** tool performs line-by-line frequency-profiling; it reports how many times each line of *source code* (q.v.) is executed.

See the **prof**(CP) and **lprof**(CP) manual pages for specific information.

Program

A program is a sequence of instructions or commands that direct a computer to perform tasks. The term "program" can refer to an *executable file* (q.v.) as well as uncompiled *source code* (q.v.). A subprogram is part of a larger program.

prot.h header file

The `<prot.h>` header file defines constants and structures for the databases that control access to the *trusted facilities* (q.v.). It should be included after the `<sys/unistd.h>` header file in the source code for any program that uses the *trusted facilities* (q.v.).

prototypes.h header file

The `<prototypes.h>` header file specifies routines that are only AT&T SVID Issue 2 compliant or added by SCO; only functions that are withdrawn from XPG3 are declared in this header file. In addition, this header file provides information about other header files and their respective functions.

See also "Header file", "a.out.h header file" and "COFF (Common Object File Format)" in this *Encyclopedia* and the manual page for **Intro**(S).

Pseudo terminals (pty's)

Pseudo-terminals are used to emulate a terminal in cases where no actual terminal device exists, such as for a window on a X Window system. A pseudo-terminal is a pair of devices, a master and a slave. This release's implementation uses *STREAMS* (q.v.) modules for communications between the master and slave portions. For more information about how pty's are implemented and examples of code that uses them, see the "Terminal Modes" chapter in W. Richard Stevens, *UNIX Network Programming*.

pwd.h header file

The `<pwd.h>` header file defines the `passwd` structure used for the *passwd* file (q.v.). See also the *passwd*(FP) manual page.

Q

qsort

qsort is a routine in the standard C library that sorts a list of items using the quicksort, or partition exchange, sorting algorithm. For information on using **qsort** see the **qsort(S)** manual page.

queue_t data structure

The **queue_t** data structure is the central point of a *STREAMS* (q.v.) module and driver. Each *STREAMS* module and driver contains two **queue_t** structures, one for each data flow direction. A **queue_t** structure contains pointers to the members of the linked list.

The **queue_t** structure is defined in the `<sys/stream.h>` header file.

Quit signal

The quit signal is one of the *signals* (q.v.) that can be generated from the keyboard. The default action for this signal is for the process to create a *core file* (q.v.) and terminate. This signal is generated when the quit character is typed at the keyboard, provided that keyboard signal generation is enabled. By default the quit character is `<Ctrl>-\`.

R

Random-number generation routines

The random-number generation routines generate pseudo-random numbers. The routines described by **rand(S)** provide simple random-number generation. The routines described by **drand48(S)** are more sophisticated and use 48-bit arithmetic.

The following functions generate pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

Random number functions

Routine	Description
drand48()	calculates non-negative double-precision floating-point values uniformly distributed over the interval [0.0,1.0); stores the last 48-bit X_i generated in an internal buffer; must be initialized
erand48(xsubi)	calculates non-negative double-precision floating-point values uniformly distributed over the interval [0.0,1.0); requires calling program to provide storage (<i>xsubi</i>) for the X_i values; does not need to be initialized
jrand48(xsubi)	calculates signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$; requires calling program to provide storage (<i>xsubi</i>) for the X_i values; does not need to be initialized
lcong48(param)	initialization entry point;
lrand48()	calculates non-negative long integers uniformly distributed over the interval $[0, 2^{31})$; stores the last 48-bit X_i generated in an internal buffer; must be initialized
mrand48()	calculates signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$; store the last 48-bit X_i generated in an internal buffer; must be initialized
nrand48(xsubi)	calculates non-negative long integers uniformly distributed over the interval $[0, 2^{31})$; requires calling program to provide storage (<i>xsubi</i>) for the X_i values; does not need to be initialized
seed48	initialization entry point that sets the value of X_i to the 48-bit value specified in the argument array
srand48(seedval)	initialization entry point that sets the high-order 32 bits of X_i to the 32 bits contained in its argument

Raw I/O

Raw I/O has two distinct meanings:

- For *device drivers* (q.v.) that support both block and character access (such as disk drivers), “raw I/O” is sometimes used to reference the character access provided. The I/O is raw in that it does not go through the system buffer cache.
- For terminal devices, “raw I/O” refers to data transfers without *canonical processing* (q.v.) applied.

Real user ID and real group ID

An active process has a real user ID and a real group ID that are the same as the user ID and group ID, respectively, of the user responsible for the creation of the process. This contrasts with the *effective UID (EUID)* and *effective GID (EGID)* (q.v.)

See also `id(C)`, `getuid(S)`, and `logname(C)` for more information.

Record

A record is an arrangement of related data within a *file* (q.v.). For example, in a file that stores address information, each record might contain data for name, address and phone number.

Under UNIX operating systems, a file is considered to be an unformatted stream of *bytes* (q.v.); records are not recognized by the operating system. Applications, however, can impose their own record structure on a file. For example, text files consist of lines of text separated by the new-line character; each line could be treated as a record.

Traditional record handling is provided on SCO UNIX system V through the SCO ISAM package. When you use SCO ISAM, a record is defined as one set of data values for all the *fields* (q.v.) that describe a specific subject. Consequently, each record in an *ISAM file* (q.v.) has a fixed length that is the sum of the lengths of all the fields in that record.

Record locking

See “File and record locking.”

regex.h header file

The `<regex.h>` header file specifies constants, macros, variables and routines used with the *regular expression* (q.v.) compiling and matching routines.

See also “Header files” in this *Encyclopedia* and the manual page for `regex(3)`.

Region data structure

The **region** data structure describes an area of memory allocated to a *process* (q.v.). It describes the size of this area and contains pointers to the *page* (q.v.) tables and the location of *swap space* (q.v.) information for associated data. For more information see Maurice Bach, *The Design of the UNIX Operating System*.

Regular expression

Regular expressions are a notation for describing patterns of characters. A regular expression is formed by concatenating characters and, usually, specific operators. These operators are sometimes called “special characters” or “meta-characters” and include “*”, “+”, and several others shown in the examples below. Regular expressions are used by *programs* (q.v.) that scan for patterns of characters in their input; some programs that support regular expressions are `awk`, `ed`, `grep`, and `lex`.

The discussion which follows describes the range of regular expressions that are available. Individual applications support different subsets of these expressions. In general, all applications which support a particular special character use it in an identical way. Some applications allow the special characters to operate only on single-character strings, while others allow special characters to operate on multi-character strings. Users should consult the documentation for the individual applications for more information on the set of regular expressions supported.

Regular expressions are summarized in the following list:

- A string of text characters with no operators at all just matches the literal string. To match the word *orange*, use:

`orange`

- The circumflex, “^”, followed by a regular expression, indicates that the expression must match a string that appears at the beginning of a line. The following expression matches the word *Golf* at the beginning of a line.

`^Golf`

- The dollar sign, "\$", appended to a regular expression indicates that the expression must match a string that appears at the end of a line. The following expression matches the word *terminal* at the end of a line:

```
terminal$
```

- An expression, followed by the "*" operator, matches 0 or more occurrences of that expression. To match a string containing any number of m's, or the null string, use the expression:

```
m*
```

- An expression, followed by the "+" operator, matches one or more occurrences of that expression. To match a string containing one or more m's, but not the null string, use the expression:

```
m+
```

- An expression, followed by the "?" operator, matches 0 or 1 occurrence(s) of that expression. This is equivalent to saying that the expression is optional. To match one occurrence of the letter "m", or the null string, use the expression:

```
m?
```

- The period character, ".", matches any single character. To match any five-letter string starting with "m" and ending with "y", use the expression:

```
m...y
```

- Alternation in regular expressions is supported using the vertical bar, "|". To match either of the strings *truck* and *fighter*, use the expression:

```
truck|fighter
```

Note that, since concatenation has precedence over alternation, the example does not match the pattern *truckfighter* or *trucfighter*.

- Expressions can be grouped using parentheses, "(" and ")". To match the strings *firetruck* and *firefighter* use the expression:

```
fire(truck|fighter)
```

NOTE Some programs such as *ed*, *sed* and *vi* use *(* and *)* for this purpose.

- Some programs permit subexpressions enclosed in parentheses or *(* and *)* to be referred to later in the regular expression. In general, the *n*th expression enclosed in parentheses can be referred to using the sequence *\n*. For example, the following expression in a *sed* program will match the word *beri-beri*:

```
\(beri\)-\1
```

- To indicate that a regular expression should be matched a specific number of times, follow that expression with a number enclosed in curly braces, "{" and "}". For example, in `lex(CP)`, to match three repetitions of `cd`, that is, `cdcdcd`, use the expression:

```
(cd){3}
```

- To specify a range of repetitions, follow the expression by two numbers, separated by a comma and enclosed in curly braces. For example, in `lex(CP)`, to match three, four, or five repetitions of `ab`, that is, `ababab`, `abababab`, or `ababababab`, use the expression:

```
(ab){3,5}
```

- A sequence of characters inside square brackets, "[" and "]", matches any one character in the sequence. To match any one of "d", "g", "k", and "a", use the expression:

```
[dgka]
```

If the circumflex, "^", is the first character inside the square brackets, then the pattern matches any character that does not appear inside the brackets. In this context, the circumflex does not signify the start of a line, as it does when prepended to a pattern. To match any character other than "a", "b", and "c", use the expression:

```
[^abc]
```

- Ranges within a standard alphabetic or numeric order are indicated with a hyphen, "-". The following expression matches any digit, uppercase letter, or lowercase letter.

```
[0-9A-Za-z]
```

- Regular expressions can be concatenated. (In fact, character strings are the result of concatenating regular expressions, the individual characters.) The resulting expression matches whatever the first expression matches followed by whatever the second expression matches. The following regular expression matches a letter followed by zero or more letters or digits.

```
[a-zA-Z][0-9a-zA-Z]*
```

- The sequences "<" and ">" indicate that a regular expression should match at the beginning or end of a word respectively. The following expression matches the string *bread* at the beginning of a word:

```
\<bread
```

- To treat an otherwise special character as a literal character, rather than as a special character, precede the character with a backslash "\". The following expression could be used to match an asterisk followed by one or more digits.

```
\*[0-9]+
```

To recognize a backslash itself, use the expression:

```
\\
```



reloc.h header file

The `<reloc.h>` header file specifies constants, macros and structures used to describe the format for COFF (q.v.) `.text` and `.data` section relocation data.

See also “a.out.h header file” and “Header files,” in this *Encyclopedia*.

S

Scancodes

Scancodes are unique values associated with pressing and releasing a key on the terminal keyboard. The routines in the scancode library provide a standard interface for writing applications that pass scancodes as well as keytop values from terminals to other scancode applications. The terminals passing scancode and keytop values must support a PC-compatible mode, and allow windowing systems to pass scancodes. The scancode functions provide user-level support that emulates the kernel `mapkey(K)` and `mapchan(K)` functions. Scancode mapping is controlled by the `ioctl(S)` commands documented on the *scancode(HW)* manual page.

Applications that use the scancode routines must include the *scancode.h* header file and be compiled with the `-lsc_s` option to the `cc(CP)` command.

The following table shows the manual pages that document the scancode interface:

Routine	Manual page	Description
keyboard	keyboard(HW)	mapping tables for scancodes
scancode	scancode(HW)	line discipline <code>ioctl</code> calls that control scancode settings on the device driver side
scanon	scanon(M)	scripts to turn the PC-scancode mode on and off
sc_copyscstate	sc_raw(S)	copies the scancode state from one tty to another
sc_exit	sc_init(S)	restores scancode terminal mode and tty settings
sc_getfkeystr	sc_init(S)	gets scancode key string
sc_getinfo	sc_raw(S)	returns scancode information
sc_getkbmap	sc_init(S)	returns pointer to scancode <code>sc_bitmap</code> structure
sc_getkeymap	sc_init(S)	returns the current active scancode keymap table
sc_getled	sc_init(S)	gets current scancode keyboard LED state
sc_getscreenswitch	sc_raw(S)	gets scancode screen switch keys
sc_init	sc_init(S)	initializes for scancode input
sc_kb2mapcode	sc_readkb(S)	maintains the state of the scancode keyboard
sc_mapcode2kb	sc_readkb(S)	gets scancode from mapcode
sc_mapcode2str	sc_readkb(S)	gets scancode map string

(Continued on next page)

(Continued)

Routine	Manual page	Description
sc_mapin	sc_readkb(S)	initializes scancode translation tables
sc_mapinit	sc_init(S)	saves for scancode mapping functions
sc_mapout	sc_readkb(S)	gets scancode mapped output string
sc_raw	sc_raw(S)	line discipline and screen-switching functions
sc_readkb	sc_readkb(S)	translation functions
sc_readmapcode	sc_readkb(S)	returns the next input mapcode
sc_readstr	sc_readkb(S)	read scancode character(s)
sc_receive_kb	sc_init(S)	maintains scancode sc_bitmap structure
sc_setfkeystr	sc_init(S)	sets scancode key string
sc_setinfo	sc_raw(S)	sets scancode information from input parameter
sc_setkeymap	sc_init(S)	sets the current active scancode keymap table
sc_setled	sc_init(S)	sets the scancode keyboard LED state
sc_setscreenswitch	sc_raw(S)	sets scancode screen switch keys
sc_str2kb	sc_readkb(S)	gets scancode keytop string
sc_unraw	sc_raw(S)	turns on scancode translation and returns the original mode

See the manual pages for more information about implementing scancodes.

SCCS (Source Code Control System)

The Source Code Control System (SCCS) is a change-tracking tool that runs under the UNIX System. SCCS is given custody of a file. When a change (or "delta") is made to the file, SCCS records who made the change and when, and can store a short comment on the change. The change is stored in the SCCS file in a way that allows the current or any previous version to be reconstructed, and the change-tracking information to be retrieved. SCCS stores the change using a technique designed to conserve disk space.

The SCCS system uses the following commands:

Command	Description
admin	create and administer SCCS control files
cdc	change delta commentary
comb	combine deltas
delta	make a delta (change) to an SCCS file
get	get a version of an SCCS file for editing or viewing
help	explain use of an SCCS command or message
prs	print information about deltas stored in an SCCS control file
rmddl	remove delta
sact	print current file editing activity
sccsdiff	compare two versions of an SCCS file
unget	undo a previous get of an SCCS file
val	validate an SCCS control file

The *sccsfile*(FP) page describes the structure of SCCS control files.

For further information on the features of SCCS, see “Source code control system (SCCS)” in the *Programming Tools Guide* and the appropriate manual pages.

sccs.h header file

The *sccs.h* header file specifies macros used to store an SCCS (q.v.) ID in a C compiled object file.

See also “COFF (Common Object File Format)” and “Header files” in this *Encyclopedia*.

Scheduling

The operating system varies the priorities of executing processes according to a specific algorithm, which is described in Maurice Bach’s *Design of the UNIX Operating System*. For instance, interactive processes gravitate towards higher priorities (because their actual runtimes are relatively small compared to their wait times), and processes that recently consumed a large amount of the CPU are relegated to lower priorities. A process’s only control over its priority is with the *nice*(S) system call and *nice*(C) command, which normally lower the relative priority of a process. If issued by the super user, these commands can grant the process a more favorable priority.

The kernel allocates the CPU to a process for a time quantum, which is determined by the *MAXSLICE* tunable kernel parameter. If the process has not released the CPU at the end of that time quantum, it is preempted and returned to the run queue at the same priority. When the kernel again allocates the CPU to the process, the process resumes execution from the point suspended. Once every second, the kernel goes through the scheduling algorithm for all processes although the priorities for all processes may not change.

scnhdr.h header file

The *<scnhdr.h>* header file specifies constants and structures used to define the section headers of a COFF (*Common Object File Format*) (q.v.) file.

See also “a.out.h header file” and “Header files” in this *Encyclopedia*.

SCSI (Small Computer Systems Interface)

SCSI (Small Computer Systems Interface) ¹ provides computers with device independence within each “class” of devices. Refer to the section “SCSI device classes” later in this article for a list of classes. SCO UNIX System V/386 contains SCSI software that consists of a *host adapter* (q.v.) driver, a disk driver, a tape driver, and kernel routines. The SCSI kernel routines provide a vehicle for creating and maintaining SCSI *device drivers* (q.v.) and host adapter drivers. The provided host adapter driver contains an optional I/O control command (*ioctl*) (q.v.) interface that permits pass-through control to the hardware on a SCSI bus.

SCSI device classes

You can add the following device classes to a host computer without modifying the “generic” system hardware or software.

Device classes

Device	Description
CD-ROM	SCSI cartridge disk with read-only memory
Communications	SCSI communications device
Disk	SCSI disk drive or a SCSI “Bootable rootable” hard disk drive
Media Changer	SCSI robot device for changing storage memory
Optical	SCSI optical memory device
Processor	SCSI processor device
Scanner	SCSI scanner device
Tape	SCSI tape drive
WORM	SCSI Write-Once-Read-Many drive

Differences between SCSI and other devices

The primary differences between SCSI devices and other devices are:

- SCSI devices have the device controller on the device itself. Non-SCSI devices have the controller built into a card in the computer; the card only knows about the device to which it is connected. With the controller on each SCSI device, the card connecting the SCSI bus to the computer can have many types of devices attached.
- SCSI devices do not generate interrupts directly, so SCSI drivers do not contain traditional interrupt handling routines. On a conventional device, an *interrupt* (q.v.) occurs whenever data is required to be read from or written to the device. A SCSI device handles I/O by informing the host adapter of its needs and the host adapter then generates an interrupt.

1. SCSI is defined by the American National Standards Institute (ANSI) (q.v.).

- SCSI disks do not require bad track scans. The controller on the SCSI disk contains its own map of tracks on the disk that are unusable. Therefore, when a SCSI disk is formatted, the controller uses its own table. This saves considerable time from either entering the information written on the disk itself or by having the administrative software find the bad tracks by examining the disk.

SCSI device configuration

Potentially, up to seven SCSI device controllers can be on a bus. Each controller can address eight peripheral devices. This means each bus can have up to 56 devices. Two buses can be provided, for a total of 112 devices. However, some host adapters impose restrictions on the actual number of devices that can be connected to a single bus.

A priority system awards interface control to the highest priority device, if multiple devices are contending for the bus. Communication on the SCSI bus is only allowed between the initiating device ² (the host adapter) and the target device.

sdevice file

The `/etc/conf/cf.d/sdevice` file contains an entry for each *device driver* (q.v.) to be included when the operating system is built. The *sdevice* file contains a keyword that correlates the information in each entry to the other device driver information files used by the operating system. When a driver described in an entry in this file controls hardware, the information required to write the entry comes from information about the hardware itself. Get the information from the hardware designer, the vendor, or possibly from a reference manual supplied with the hardware.

When the operating system is built, the *link kit* (q.v.) searches *sdevice* for the drivers that are to be included. An entry in this file specifies whether the driver should be included or excluded. If a driver is to be included, an entry of the same name needs to be included in the *mdevice file* (q.v.) describing the features of the device driver.

Further information describing each of the fields for an *sdevice* entry can be found on the *sdevice(F)* manual page.

2. Although SCSI protocol provides for the connection of multiple initiators (devices that can initiate an operation) and multiple targets, SCO System V/386 software assigns the host adapter to be the initiator, and only supports a single-initiator and multiple-target arrangement.

search.h header file

The <search.h> header file specifies type definitions and routines used by the library search routines.

See also “Header files” in this *Encyclopedia* and the manual pages for `bsearch(S)`, `hsearch(S)`, `lsearch(S)`, and `tsearch(S)`.

Section

A section is the smallest portion of a COFF (*Common Object File Format*) (q.v.) or OMF (*Object Module Format*) (q.v.) object file that is relocated and treated as one separate and distinct entity. In the most common case, an object file contains three sections named `.text`, `.data`, and `.bss`. Additional sections accommodate multiple text, or data segments, shared libraries, shared data segments, or user-specified sections.

NOTE Do not assume that every COFF file has a certain number of sections, and do not assume characteristics of sections such as their order, their location in the object file, or the address at which they are to be loaded. This information is available only after the object file has been created. Programs manipulating COFF files should obtain that information from the file and section headers in the file.

Security

System security refers to protections provided against unauthorized access to system resources, especially sensitive data. The discussion here pertains only to software issues; a full discussion of security also would include physical issues such as access to the equipment itself, whether the system can be accessed over phone lines, and what type of cabling is used for the system.

The *Department of Defense Trusted Computer System Evaluation Criteria*, known as the “Orange Book,” is published by the United States government and specifies levels of security for computer systems. We will use their terminology for this discussion. The Orange Book uses A, B, C, and so forth, to represent decreasing classes of security, and appends an Arabic number to the class (1, 2, 3, and so forth) to represent increasing security within a class. For example, level B is more secure than level C, but C2 is more secure than C1. UNIX System V/386 supports C1 and C2 security.

C1 security

Virtually all XENIX and UNIX operating systems are capable of supporting C1 security, which specifies a separation of users and data. Each user has private disk space and the ability to specify access to each file for the owner, group, and all others. The super user overrides all security limitations, and can

access any resource on the system. See the “Permissions, file” article for more information.

C2 security

C2 security requires more fine-grained discretionary protection, resource isolation, audit capability, and authentication enhancements. The *Trusted Facilities* (q.v.) included with Release 3.2 of SCO UNIX System V provides this level of security with the following features:

- LUID 's (logged-in user identifications) that enhance accountability by associating users with processes. A child process inherits the LUID of its parent process.
- resource isolation that prevents a user from accessing resources such as memory, disk space, and commands without proper authorization
- script for setting and checking file modes for a file's owner, group, and so on
- privileges that are associated with individual commands and are checked before the command is executed
- audit records that can be produced by any program written for the system. Kernel modules have been modified to produce audit records. An audit daemon reads, collects, and compresses these records for an audit program that reports on all events that could represent attempted security breaches.
- encrypted passwords that are hidden from users
- authentication data is protected from users
- ability to control whether users generate their own passwords
- User Authentication Database that specifies the minimum login change time, the number of allowable login attempts, the expiration date of a password, command privileges for the user, and so forth. A large number of unsuccessful logins from a terminal may indicate an unauthorized user attempting to break security; the User Authentication Database provides information that can be used to identify when a terminal should be locked.

See the “Trusted facilities” article for more detailed information about implementing trusted applications on UNIX System V/386.

security.h header file

The `<sys/security.h>` header file contains general purpose defines and macros for the *trusted facilities* (q.v.). It should be included in the source code for any program that is part of a trusted application, somewhere after the `<unistd.h>` header file.

Semaphores

A semaphore is an atomic synchronization mechanism. Semaphores provide a mechanism that can be used to arbitrate the use of resources among processes and to synchronize processes with external events.

The 3.2.4 Development System supports two separate semaphore implementations, commonly referred to as “System V semaphores” and “XENIX semaphores.” These are each discussed in separate articles.

Semaphores (System V)

System V semaphores are one of the *IPC facilities* (q.v.). The following table lists the system calls used for working with semaphores; see the reference pages in section (S) for further information about using each.

System V Semaphoring system calls

System call	Description
<i>semid = semget(key, nsems, semflg)</i>	creates semaphore array, identified by <i>semid</i> , which is associated with <i>key</i> . The number of semaphores in the array is controlled by <i>nsems</i> . <i>semflg</i> controls access permissions to the array and determines whether a new semaphore array should be created or the <i>semid</i> of an existing semaphore array returned.
<i>semop (semid, sops, nsops)</i>	increments, decrements, or checks the value of the semaphores described by the <i>sops</i> array.
<i>semctl (semid, semnum, cmd, arg)</i>	performs various control operations as specified by <i>cmd</i>

See the manual pages, the “Advanced Interprocess Communication” chapter in Marc Rochkind, *Advanced UNIX Programming* and the “Interprocess Communication” chapter in W. Richard Stevens, *UNIX Network Programming* for more information about using System V semaphores.

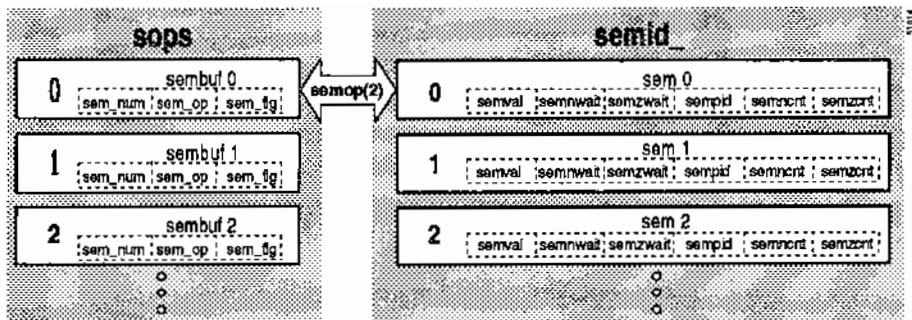
System V semaphore structures

Each UNIX semaphore includes a counter that typically describes the number of resources available for use. When the value of the semaphore is 0, the resource is unavailable; the process can either wait for the value of the semaphore to be incremented (indicating that the resource has become available) or return without accessing the resource.

A semaphore is represented by a data structure (**sem**) that contains the following members:

semval	value of the semaphore
sempid	process identifier (PID) of the last process that performed an operation on the semaphore
semncnt	number of processes waiting for the semaphore value to be incremented
semzcnt	number of processes waiting for the semaphore value to be equal to zero

The array of semaphores is identified by **semid**; each semaphore in the array is identified by an integral number, beginning with 0, and has a **sem** structure associated with it. Each process creates a local structure (**sops**) that is tied to the semaphore number, and the process uses this structure with the **semop(S)** call to increment and decrement the semaphore. A process may create the local **sops** structure "on the fly"; the same **sops** structure can be used to reference different semaphores at different times. The following diagram illustrates the relationship of the structures used with semaphores.



UNIX System V semaphore structures

The operating system ensures that only one process manipulates a semaphore set at any given time. Simultaneous requests are performed sequentially in an arbitrary manner. No operations are performed on any semaphores in a set until all operations are successful.

Any unsuccessful nonblocking operation for a single semaphore or a set of semaphores causes immediate return with no operations performed at all. When this occurs, a known error code (-1) is returned to the process and the external variable **errno** is set accordingly.

The blocking semaphore operation allows processes to control their execution based on the values of semaphores at different points in time.

An array of these blocking/nonblocking operations can be performed atomically on a set containing more than one semaphore:

- The blocking/nonblocking operations can be applied to any or all of the semaphores in the set when performing an array of operations.
- The operations can be applied in any order of semaphore numbers. However, no operations are done until they all can be done successfully.
- The `SEM_UNDO` flag can be set to allow the operation to be “undone” at process termination, as a cleanup activity. Undoing is accomplished by using an array of “adjust values” for the operations that are to be undone.

Tunable parameters for System V semaphores

Tunable parameters control the size of the kernel structures used for System V semaphores as well as other characteristics. When you use System V semaphores in an application, the system administrator may need to adjust these tunable parameters to appropriate values.

The tunables are:

- SEMMAP** determines the number of entries in the map that controls semaphore arrays. Each initialized semaphore array requires one entry in this map. The value of `SEMMAP` must be greater than or equal to `SEMMNI`.
- SEMMNI** sets the maximum number of semaphore arrays that can exist on the system
- SEMMNS** sets the maximum number of semaphores that can exist on all semaphore arrays in the system. This number is usually set significantly lower than `SEMMNI*SEMMSL`, because most environments do not use many large semaphore arrays.
- SEMMNU** number of “undo” structures in the system. Undo structures are used to support the `SEM_UNDO` error recovery mechanism.
- SEMMSL** maximum number of semaphores in each semaphore array
- SEMOPM** maximum number of semaphore operations. This value determines the largest value that the *nsops* argument to `semop(S)` may take.
- SEMUME** maximum number of entries allowed in an undo structure. The value of `SEMUME` must be greater than or equal to `(SEMMSL -1)`.
- SEMVMX** maximum value a semaphore can have. This limits the value that the `semval` member of the `sems` structure can have.
- SEMAEM** maximum adjustment to semaphore value made when the process exits. Customers should never modify this value.

If a process issues a call that would exceed one of these values, the call fails. Use the `sar -m` report to gauge whether the current values are appropriate for your environment.

Semaphores (XENIX)

The first releases of the XENIX operating system were ported from an AT&T operating system that did not support semaphores, so the XENIX semaphore mechanism was developed. This mechanism is supported on UNIX System V/386 for code that is compiled with the `-lx` option. The following table lists the functions used for the XENIX semaphore mechanism.

XENIX Semaphoring functions

Function	Description
<code>creatsem(sem_name, mode)</code>	create semaphore; return semaphore number, <i>sem_num</i> , that can be opened and used in subsequent semaphore operations. <i>mode</i> determines accessibility to the semaphore using the same format as file access bits.
<code>opensem(sem_name)</code>	open <i>sem_name</i> semaphore for use by the issuing process
<code>waitsem(sem_num)</code>	request control of <i>sem_num</i> semaphore. If the semaphore is available, control is given immediately; otherwise, the process waits.
<code>nbwaitsem(sem_num)</code>	like <code>waitsem</code> , but returns error if the semaphore is not available rather than blocking the process until the semaphore is available
<code>sigsem(sem_num)</code>	relinquish control of the <i>sem_num</i> semaphore and signal this fact to any processes that are waiting for the semaphore

The XENIX semaphore mechanism uses a file inode rather than internal kernel data structures.

See the “Advanced Interprocess Communication” chapter in Marc Rochkind, *Advanced UNIX Programming* for more information about using XENIX semaphores.

setjmp.h header file

The `<setjmp.h>` header file specifies constants, type definitions and routines used for nonlocal transfer of control in a C program.

See also “Header files” in this *Encyclopedia* and the manual pages for `setjmp(S)` and `longjmp(S)`.

SGML (Standard Generalized Markup Language)

The Standard Generalized Markup Language (SGML) is a standard for document interchange, adopted by the International Organization for Standardization (ISO) (q.v.) and codified in the ISO Standard 8879 (1986). SGML is used for describing structured information, usually documents that are being processed by a typesetting or database system. Documents marked up in SGML are easily parsable and are encoded in a way that is not dependent on a particular software or hardware system.

Traditionally, markup, such as typesetting codes, has been used to describe the form of a document, that is, its physical appearance. SGML provides a notation that describes documents by describing their content. SGML is used to designate the structural elements of a document, such as "title," "chapter," and "paragraph," and refers as little as possible to the physical appearance.

All SGML markup consists of ordinary text characters, which delimit structural elements, make reference to pre-defined text or files, and include non-textual information pertaining to structural elements. Every SGML document is accompanied by a document type definition (DTD), which is a form of grammar defining the structure of the document. A DTD will typically be used to describe many documents having the same structure. There are a number of widely-used, publicly-available DTDs, such as the Association of American Publishers (AAP) DTD for books and articles.

The standard reference on SGML is ISO Standard 8879. The most thorough discussion of SGML currently available is *The SGML Handbook*, by Dr. Charles Goldfarb, editor of ISO 8879 (Oxford University Press, 1990). This book also contains the full text of the Standard. Other useful references are *SGML: An Author's Guide* by Martin Bryan (Addison-Wesley, 1988), and *Practical SGML* by Eric van Herwijnen (Kluwer Academic Publishers, 1990).

There are a number of commercial and public-domain products for facilitating the preparation of SGML documents and DTDs.

See also "HyTime" and "DSSSL" in this *Encyclopedia*.

sgtty.h header file

The `<sgtty.h>` header file specifies constants used for backward compatibility. The `g` and `s` system calls have been superseded on UNIX systems, but are still supported. The `<termios.h>` header file (q.v.) and the `<sys/termio.h>` header file (q.v.) provide all the functionality of the `<sgtty.h>` header file in a standards conformant manner for UNIX systems.

See also "Header files" in this *Encyclopedia*.

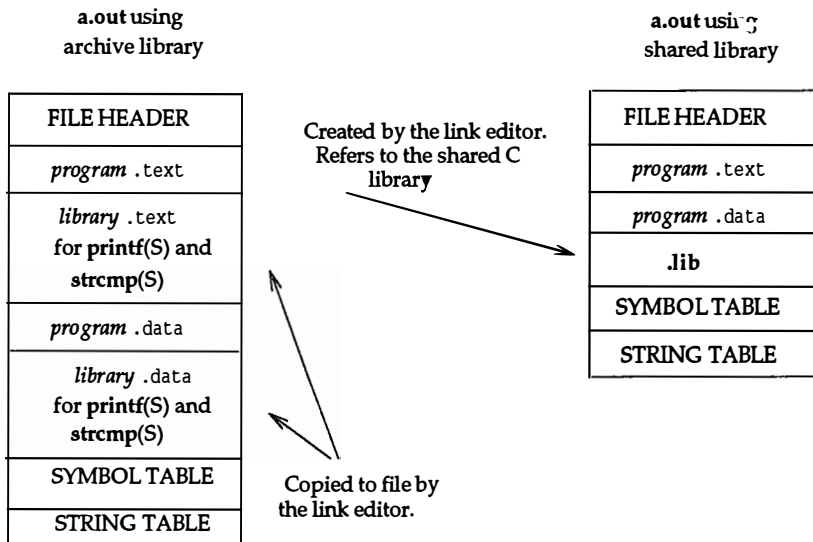
Shared libraries

A shared library is a file containing object code that several processes may use simultaneously while executing. When a program is link edited with a shared library, the library code that defines the program's external references is not copied into the program's object file. Instead, a special section called *.lib* that identifies the library code is created in the object file. When the resulting *a.out* file is executed, it uses the information in this section to bring the required shared library code into the address space of the process. It should be noted that only one copy of the shared library code is in RAM and is shared by all processes using that library.

To understand how shared libraries work, consider compiling the following simple program:

```
main()
{
    ...
    printf( "How do you like this manual?\n" );
    ...
    result = strcmp( "I do.", answer );
    ...
}
```

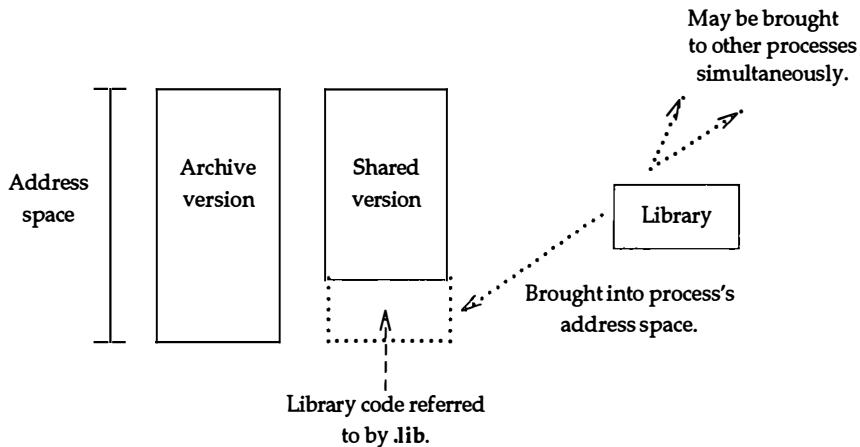
The **printf(S)** and **strcmp(S)** routines are in the standard C library, which can be accessed either as an archive or a shared library. The following diagram illustrates the *a.out* files created using the archive library versus the *a.out* files created using the shared library.



a.out files created using an archive library and a shared library

When using a shared library the virtual memory address of the symbols are fixed at the link editing stage. This shortens the load time of a program. This contrasts with the methods used by both *dynamic libraries* (q.v.) and *archive libraries* (q.v.) to resolve external references.

When several *a.out* files need the same code from an archive library, each file gets its own copy of the code. This results in duplication of the same code on the disk and in memory when the *a.out* files are run as processes. In contrast, when a shared library is used, the library code remains separate from the code in the *a.out* files, as indicated in the following diagram. This separation enables all processes using the same shared library to reference a single copy of the code.



Processes using an archive and a shared library

Tools used with shared libraries

The following table summarizes the tools that are available for work with shared libraries. See the appropriate manual pages and the "Archives and Shared Libraries" chapter in the *Programming Tools Guide*.

Tools for shared libraries

Command	Description
<code>mkshlib(CP)</code>	creates a shared library
<code>chkshlib(CP)</code>	checks for compatibility between shared libraries
<code>dump -hv a.out</code>	<code>dump(CP)</code> determines whether the <i>a.out</i> file uses shared libraries by determining whether it includes a <i>.lib</i> section
<code>dump -L a.out</code>	displays the libraries used in the <i>a.out</i> file

Deciding whether to use a shared library

Consider the following criteria when deciding whether your application would benefit from making your library shared or static (archived):

- A well-designed shared library almost always saves space, because shared library code is not copied into the *a.out* files that call it.
- Using shared library routines may reduce the dynamic memory needs of processes because library code is shared at runtime. However, if the program uses only a small part of a shared library's text and data, executing the program might require more memory with a shared library than without it, because each process gets its own copy of the entire shared library's data region.
- Code and data references between the program and the library may be slower than for archive libraries.
- Shared libraries may make the code easier to maintain. Because shared library code is brought into a process's address space at runtime, updating a shared library effectively updates all programs that use the library. If an error in shared library code is fixed, all processes automatically use the corrected code.

Note that, in some cases, this may be a disadvantage when it causes a program to behave differently even though the code has not been modified or recompiled.

- Each program can call a limited number of shared libraries. The number of libraries that can be called is controlled by the `SHLBMAX` tunable parameter, which can be set to a value between 3 and 6. Adherence to this limit is checked at execution time, not when the program is compiled.
- Shared libraries may be more difficult to build and maintain than archive libraries.
- Shared libraries are not available on the XENIX operating system or UNIX System V releases prior to Release 3.0. If you need to port your program to such systems, use archive libraries.

Shared memory

Cooperating processes map the same memory region into their address spaces. All the processes that use this shared memory segment can write to and read from that memory region. In many applications, a separate interprocess communication mechanism (usually a semaphore) is used to coordinate access to the shared memory region. Otherwise, one process may be in the middle of updating the information when another process attempts to read it, possibly resulting in data corruption.

All SCO UNIX systems and XENIX, Release 2.3 Version 4, support two systems of shared memory, called UNIX System V shared memory (or IPC shared memory) and XENIX shared memory (or shared data), respectively.

Shared memory, System V

Shared memory is one of the *IPC facilities* (q.v.) for interprocess communication. Shared memory is the fastest interprocess communication mechanism.

Using System V shared memory

The following table lists the library routines used for UNIX shared memory.

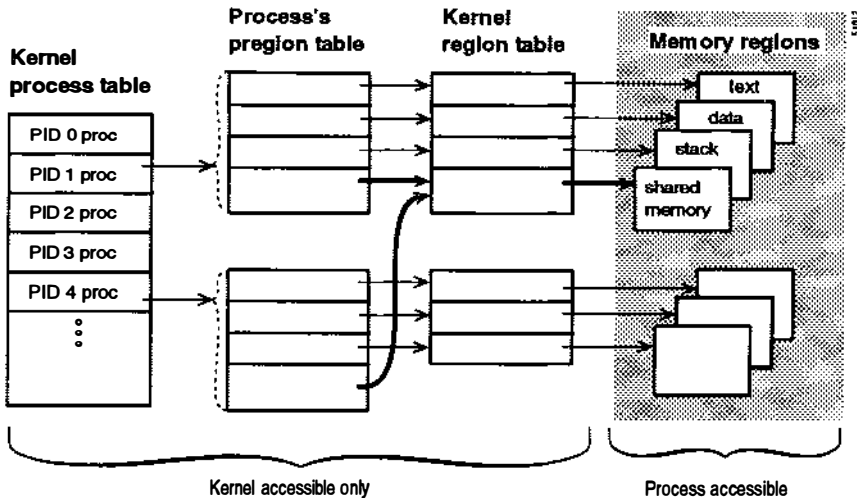
System V shared memory routines

Routine	Description
<code>shmget(key, size, shmflg)</code>	creates shared memory region or get identifier for existing region
<code>shmat(shmid, shmaddr, shmflg)</code>	attaches shared memory region
<code>shmdt(shmaddr)</code>	detaches shared memory region
<code>shmctl(shmid, cmd, buf)</code>	shared memory control operations

For more information about using UNIX shared memory, see the `shmget(S)`, `shmop(S)`, and `shmctl(S)` manual pages and the “Advanced Interprocess Communications” chapter in Maurice Bach’s *Design of the UNIX Operating System*.

System V shared memory structures

The following diagram illustrates how the shared memory region is mapped into process’s address space. When the owner process creates the shared memory region with the `shmget(S)` system call, the process’s `preigion` table points to the kernel’s region table, which then points to the shared memory region. When another process attaches the shared memory region with the `shmat(S)` system call, the entry in that process’s `preigion` table points to the same entry in the kernel region table as the original process; this enables both processes to access the same memory segment. When a process exits, its `preigion` table is released, including that pointer; if the process issues the `shmdt(S)` system call to detach from the shared memory region before it exits, its `preigion` pointer to the shared memory segment is released.



Mapping shared memory regions

The `shmid_ds` data structure serves as a header to the shared memory region. It contains control information (such as the number of processes attached to this region, the PID of the owner, the last operation, and the time of the last attach, detach, and control operation) as well as a structure that defines access permissions for the shared memory region. These structures are discussed on the *shm(FP)* manual page.

System V tunable parameters for shared memory

A set of tunable parameters controls the size and number of structures used with System V shared memory. When you install an application that uses shared memory, you may need to adjust these parameters to appropriate values.

- SHMBRK** number of pages reserved above data segment. This number is used to calculate the address where `shmat(S)` attaches the first shared memory segment when the operating system chooses an address.
- SHMALL** maximum number of shared memory regions that can exist on the system at one time
- SHMMAX** maximum size of one shared memory segment, in pages. If a `shmget(S)` call uses a larger value for the second parameter (*size*), the call fails. Note that `shmget` specifies the size in bytes, not pages.

- SHMMIN** minimum size of one shared memory segment, in pages. If a **shmget(S)** call uses a smaller value for the second parameter (*size*), the system automatically rounds up the size to the size specified here.
- SHMMNI** maximum number of shared memory segments (**shm_ds** structures) that can exist on the system at one time
- SHMSEG** maximum number of segments to which one process can attach with **shmget(S)** at one time. This value is often set fairly low.

If a process issues a call exceeding the limit set by these tunable parameters, the call fails.

Shared memory, XENIX

XENIX shared memory (also is called shared data) is available for programs compiled with the **-lx** option. Rather than using kernel structures, it creates a file and uses the file's inode to control the shared data region. This means that the shared data region has an entry in the system *inode table* (q.v.) and file descriptor table; a child process inherits these table entries.

A reference count records the number of times the segment has been accessed. When the shared data segment is created, its reference count is 0. Each time a process issues the **sdleave(S)** function to leave the shared data segment, the version number is incremented.

The following table summarizes the functions used for XENIX shared data.

Routine	Description
sdget (<i>path</i> , <i>flags</i> [, <i>size</i> , <i>mode</i>])	creates a shared data segment of <i>size</i> bytes. <i>path</i> specifies the location of the file used to control the shared data segment and <i>flags</i> defines how the segment should be created
sdenter (<i>addr</i> , <i>flags</i>)	enters the shared data section at address <i>addr</i> ; <i>flags</i> determines the type of access, such as read-only or write-only
sdleave (<i>addr</i>)	leaves the shared data section located at address <i>addr</i>
sdgetv (<i>addr</i>)	return version number of shared data section located at address <i>addr</i>
sdwaitv (<i>addr</i> , <i>vnum</i>)	wait until version number for the segment located at address <i>addr</i> is no longer equal to <i>vnum</i>
sdfree (<i>addr</i>)	detaches the shared data section located at address <i>addr</i> from the process

For more information about using XENIX shared data, see the “XENIX cross-development notes” chapter in the *Developer's Topics*.

Shells

A UNIX System shell is a command interpreter. It gets its name because it is an outer layer over the *kernel* (q.v.) and the hardware which are at the core of the UNIX operating system. A shell provides an interface between the user and the kernel. A shell is simply a program—it is not part of the operating system. Therefore, the user is not restricted to one particular shell. Currently there are three shells in widespread use. The oldest of the three is the *Bourne shell*, *sh* (q.v.). The next shell developed was the *C shell*, *cs**h* (q.v.). The latest shell to be developed is called the *Korn shell*, *ksh* (q.v.), a shell that combines the best features of the previous two.

The shell accepts instructions from the user and translates them into tasks that the kernel understands and can perform. Usually, it interprets a command and generates a corresponding *process* (q.v.). In this way, a shell allows the user to take advantage of the routines, utilities, and devices supported by the kernel.

The second main function of a shell is that it also can be used as a programming language. A program written within the shell environment is referred to as a shell program or as a shell script. Some shells provide a programming language almost as sophisticated as high-level programming languages. Shell programming includes such advanced programming concepts as functions, logical branching, control flow, and input/output specifications.

Signals

Signals are a UNIX operating system facility that inform processes of errors or asynchronous events. Some examples of events that signals monitor are:

- bad memory access
- bus error
- bad argument to a system call
- death of a child
- kill the process
- suspend/continue the process

Specific examples of signals are the **SIGHUP** signal, which is generated when a dial-in line loses its carrier signal or a controlling process dies, and the **SIGINT** signal, which is generated when some programmed “interrupt” is pressed on the keyboard.

The complete list of signals, each associated with some specific event, and their designations may be found on the **sigaction(S)**, **signal(S)**, and **sigsetv(S)** manual pages. The **sigaction** manual page discusses POSIX-compliant signal

handling, while the other two pages describe signal handling that conforms to AT&T SVID Issue 2 and iBCSe2 standards. The signals themselves are defined in the `<sys/signal.h>` header file.

Signals have one of these possible states:

- pending** Between the time the operating system dispatches a signal and when a process catches it, the signal is pending.
- caught** When the process acts on a signal, the signal is caught. A routine known as the “signal handler” will execute when a signal is received, a default action is taken, or the signal will be ignored.
- blocked** When a signal is temporarily halted from being received and stays pending until the block is removed, then the signal is blocked.
- held** (AT&T SVID Issue 2 or iBCSe2 only) Upon receipt of a signal, it is stored until later released. In this case, it is held.
- ignored** When a process is not reacting to signals, the signal is ignored.

The process has the choice to act upon or ignore a signal, except for the **SIGKILL** and **SIGSTOP** signals, which can neither be caught or ignored.

The **signal(S)** system call is provided for backward compatibility and has been superseded by the current signal functions to be described later in this article.

The table that follows lists all the signal functions. The “Stdndrd” column indicates the standard with which this function is associated, POSIX (P), AT&T SVID Issue 2 and iBCSe2 (S), or XENIX (X).

The signal functions are:

Routine	Manual Page	Stdnd	Description
sigaction	sigaction(S)	P	identify signal handling action
sigaddset	sigset(S)	P	add signal to block to signal set
sigdelset	sigset(S)	P	delete signal from signal set
sigemptyset	sigset(S)	P	get an empty signal set
sigfillset	sigset(S)	P	get a full signal set
sigismember	sigset(S)	P	tests a signal to see if in signal set
siglongjmp	sigsetjmp(S)	P	restore environment on signal
sigsetjmp	sigsetjmp(S)	P	store environment prior to signal
sigsuspend	sigsuspend(S)	P	wait for a signal
sigprocmask	sigprocmask(S)	P	block signals using signal set
sigpending	sigpending(S)	P	assign signal set to current pending and blocked signals
sighold	sigsetv(S)	S	hold signal until priority lowers
sigignore	sigsetv(S)	S	set SIG_IGN variable
sigpause	sigsetv(S)	S	suspend process until signal received
sigelse	sigsetv(S)	S	restores signal action
sigset	sigsetv(S)	S	establish action for signal
signal	signal(S)	S	handle a signal
sigsem	sigsem(S)	X	signal process waiting for semaphore

POSIX signal handling

POSIX signal handling consists one or more of these activities:

1. Creating a signal set and blocking signals before performing initialization
2. Establishing a signal action
3. Enabling signals to be caught
4. Performing an action when a signal is caught

An example of these signal manipulation activities is provided in the “Job control under ksh” chapter in the *Developer's Topics* guide. Additional POSIX signal handling information is provided in the O'Reilly & Associates *POSIX Programmer's Guide* by Donald Lewine; additional AT&T SVID Issue 2 and iBCSe2 signal handling information can be found in Prentice-Hall's *Advanced UNIX Programming* by Marc J. Rochkind.

Use of the signal set

A signal set is a group of signals that are to be manipulated in some customized way. Two choices exist for creating a set, either starting with an empty set using `sigemptyset(S)` and adding signals to the set using `sigaddset(S)`; or starting with a full set of all signals using `sigfillset(S)` and deleting signals from the set using `sigdelset(S)`. These signal sets are then used by the `sigprocmask(S)` and `sigaction(S)` routines as arguments; the `sigpending(S)` routine returns a signal set.

Establishing a signal action

The **sigaction(S)** system call is used to establish the signal handler. **sigaction** is called for each signal for which an action is to be specified. The **sigaction** system call acts upon the information in the **sigaction** structure, which is structure defined in the `<sys/signal.h>` header file. This structure is:

```
struct sigaction {
    void (*sa_handler)(); /* signal handler */
    sigset_t sa_mask;      /* signal mask */
    int sa_flags;          /* signal flags */
};
```

The values for `sa_handler` can be **SIG_DFL** to perform a default action that is defined by the operating system, **SIG_IGN** to ignore the signal, or the name of a routine, referred to as the signal handler. For example, if the signal handler is to be executed on an occurrence of one of four different signals, **sigaction** must be called four times, with each call establishing one of signals. When **sigaction** is called, it stores the old action in the third argument of the call. The old action is stored because it may be used later in the signal handler to reestablish the previous signal state.

Performing an action

When the signal handler is executed, it performs the action determined by the application. In most cases, this consists of a cleanup of the state created by the **sigaction** call and any other application-specific code.

AT&T SVID Issue 2 or iBCSe2 signal handling

The AT&T SVID Issue 2 and *Intel binary compatibility specification (iBCS)* (q.v.) issue 2 standards specify a set of signal-handling routines that were established prior to the more recently developed POSIX routines. For the most part, these routines exist only to provide backward compatibility. The **sigsetv(S)** manual page lists the functions and the possible signal values. The **sigset(S)** function is analogous to POSIX **sigaction(S)** in that **sigset** establishes a signal action of **SIG_DFL**, **SIG_IGN**, or unique to this routine, **SIG_HOLD**. **SIG_HOLD** indicates that a signal be held upon receipt until specifically released using the **sigrelse(S)** call. **SIG_HOLD** is used with the alternative action of having a signal handler routine execute when the specified signal is held. The signal handler routine must then call **sigrelse** to release the signal. The **sigignore(S)** call sets **SIG_IGN**; **sighold(S)** sets **SIG_HOLD**.

Sockets

Sockets is an application program interface (API) that provides a framework for communication between processes executing either on the same host or across a network protocol. Sockets are bidirectional, providing a two-way flow of data between processes that may or may not have the same parent.

A socket is the basic building block for communication, providing an end-point to which a name may be bound. Sockets are created within a communications domain much as files are created within a filesystem.

The following table lists the library routines supported in the *SCO TCP/IP Socket Programmer's Guide*.

Library functions

Library functions	Description
bstring(SLIB)	byte and byte string operations
byteorder(SLIB)	convert values between host and network byte order
getdtablesize(SLIB)	get descriptor table size
gethostbyname(SLIB)	get network host entry
gethostname(SLIB)	get/set name of current host
getnetent(SLIB)	get network entry
getprotent(SLIB)	get protocol entry
getservent(SLIB)	get service entry
gettimeofday(SLIB)	get date and time
inet(SLIB)	Internet address manipulation routines
insque(SLIB)	insert/remove element from a queue
killpg(SLIB)	send signal to a process group
perror(SLIB)	system error messages
rcmd(SLIB)	routines to return a stream to a remote command
resolver(SLIB)	resolver routines
rexec(SLIB)	return stream to a remote command
string(SLIB)	4.3 BSD string operations
syslog(SLIB)	control system log
wait3(SLIB)	wait for process to terminate or stop

The following table lists the system calls supported in the *SCO TCP/IP Socket Programmer's Guide*.

System calls

System call	Description
accept(SSC)	accept a connection on a socket
adjtime(SSC)	correct the time to allow for synchronization of the system clock
bind(SSC)	bind a name to a socket
connect(SSC)	initiate a connection on a socket
getpeername(SSC)	get name of connected peer
getsockname(SSC)	get socket name
getsockopt(SSC)	get and set options on sockets
listen(SSC)	listen for connections on a socket
networks(SSC)	names and numbers for the Internet
recv(SSC)	receive a message from a socket
select(SSC)	synchronous I/O multiplexing
send(SSC)	send a message to a socket
shutdown(SSC)	shut down part of a full-duplex connection
socket(SSC)	creat an endpoint for communication

For a very practical discussion of the sockets interface, see W. Richard Stevens' *UNIX Network Programming*. *UNIX Networking* by Stephen G. Kochan and Patrick H. Wood provides a more limited and theoretical look at sockets in the context of a chapter on TCP/IP.

Source code

Source code is the uncompiled version of a *program* (q.v.). Before the program can be linked and executed, the source code must be translated to machine language by a compiler. Once compiled, it is referred to as "binary" code. Sometimes source code is processed by an interpreter, which runs the code without translating it into machine language.

spl (set priority level)

A set of “spl” routines (see **spl(K)**) are provided that enable *device drivers* (q.v.) and other kernel code to enable or disable *interrupts* (q.v.) to protect critical code sections within a device driver. The actual names of the routines and the interrupts they govern for UNIX System V/386 running on Intel 80386 systems are shown as follows:

Function	Interrupt	Device or Description
spl0	0	Enables all interrupts
spl1	1	Context and process switch interrupts
spl2	2	Low-level system use
spl3	3	Low-level system use
spl4	4	Character device interrupts
spl5	5	Character device interrupts such as ports cards, and character lists (clists)
spl6	6	System clock, block devices, and network devices
spltty	7	tty devices (terminals)
spl7	8	Prevents all interrupts
splx	—	Restore interrupts to previous level

Additional **spl** functions are provided for historic purposes that duplicate the previously listed functions: **splcli** and **splpp** are the same as **spl5**, **splbuf** and **splni** are the same as **spl6**, and **splhi** is the same as **spl7**. Note that, while the use of **spl** routines is common across most UNIX systems, the actual **spl** levels and their meanings tend to vary between hardware platforms and operating system versions.

An **spl** function disables interrupts for the interrupt level specified and all interrupts of lower values. For example, the **spl5** function disables interrupts from levels 0 to 5 inclusive. See the **spl(K)** manual page for more information about these routines and an example of their usage.

spl routines should be used with caution to avoid degrading system performance. Running a section of code at a high priority for long periods delays the processing of interrupts. Furthermore, if the system *clock* (q.v.) interrupt is disabled for a time longer than a clock tick, clock accuracy degrades. For these reasons, sections of “critical code” that are protected by **spl** routines should be kept as short as possible.

The interrupt value is also known as the IPL, or interrupt priority level. This number is set in the `/etc/conf/cf.d/sdevice` file. Refer to the “sdevice file” article in this *Encyclopedia* for more information.

Stack

A stack is a data structure that provides access to items in last-in, first-out (LIFO) order. The user stack is one of the three memory regions that are created for each executing process. The user stack region is created automatically by the kernel at run time.

The stack consists of a set of logical “stack frames”; these frames are defined by the hardware architecture and by the compiler—the so-called “calling conventions” or “calling sequence”. Each subroutine call causes a new frame to be pushed, and a return causes it to be popped from the stack. The stack pointer is a hardware register that indicates the current stack frame.

Each user-level process has two stacks: the user stack that contains information used when the process is executing in user mode, and the kernel stack that contains information used when the process is executing in kernel mode. Storage for the kernel stack is in the process `u_area` (see the article “User structure”)

The kernel is not a process and does not have a separate stack. The kernel stack for a process can be viewed with the `crash->trace` and `crash->stack` commands. The kernel stack contains the stack frames for functions executing in kernel mode. The kernel stack structure is similar to the user stack of a process.

The size of the stack of a program compiled in native System V/386 mode is adjusted dynamically at run time. System V/286 binaries that are run on System V/386 have a 4 kilobyte, fixed size stack, unless a different size stack is specified to the linker with the `-F` option. Use the `-F` option to the `fixhdr(C)` command to change the stack size of an existing System V/286 binary without recompiling it.

See Maurice Bach, *Design of the UNIX Operating System*, for more information about the stack.

Standard I/O

Standard I/O refers to both: the standard input, the standard output and the standard error *file descriptors* (q.v.) and a library of high-level functions that provide enhanced performance over that of the *file access system calls* (q.v.), such as `read()` and `write()`.

Typically, the standard file descriptors are established by `getty(M)`, and inherited by subsequent processes, such as `login(M)` and `sh(C)`. In most cases, programs read their input from the standard input, write their output to the standard output, and write diagnostic information to the standard error. Unless otherwise specified, the standard input corresponds to the keyboard of the user who invokes the process while the standard output and standard error

are associated with the user's display. Usually the standard error retains its association with the display, whereas a new child process often redirects its standard input and/or output before executing a program. A common example of this redirection process is the shell, which provides users with a notation for specifying that the standard input, output, and error streams of a program be redirected before execution.

As well as the file access system calls, a library of high-level functions, referred to as "file access library routines" is provided. These functions are built on top of the low-level functions and access the standard I/O streams using the constants `stdin`, `stdout`, and `stderr`. (Other streams can be created by the process as needed.) The standard I/O constants, macros and functions are defined in the `<stdio.h>` header file (q.v.).

Programs should to use the high-level functions instead of the low-level functions for two reasons: efficiency and standards conformance. Programs that use the high-level functions may be more efficient because of the automatic buffering the functions provide. Also, programs that use the low-level functions will not comply with some standards; the ANSI specification for the C programming language, for instance, does not include the low-level functions.

Standards

Over the years, a number of variations of UNIX operating systems and compilers have evolved. While all these variants have a common ancestor, the differences between versions was, ironically, starting to threaten the portability that is one of the major advantages of the UNIX operating system. A number of attempts have been made to standardize the user-level interfaces of the operating system. Some of the major standardization efforts are the System V Interface Definition (SVID), POSIX, the Open Software Foundation (OSF), ANSI C, and X/Open Portability Guide. These standards are discussed in separate articles.

The C compiler supplied with this release of the Development System enables you to specify the standardization mode (if any) under which you want the program compiled. See the *Programming Tools Guide* and the `cc(CP)` manual page for more information.

Static linking

Static linking refers to the requirement that symbolic references be resolved at linking time. If the *link editor* (q.v.) finds an undefined symbol in an object file, then it attempts to find a definition for that symbol in another object file or in an *archive library* (q.v.). If the definition is found in a library, then the code for the definition is incorporated into the object file created by the link editor. If any symbols are unresolved, the object file is not executable. See also "dynamic linking" in this *Encyclopedia*.

stdlib.h header file

The `<stdlib.h>` header file specifies function prototypes for the routines in the standard library. To ensure correct routine behavior, include *stdlib.h* when any of these functions are used.

See also “Header file” and “Library routines” in this *Encyclopedia*.

Sticky bit

The sticky bit is used on older UNIX systems to improve the performance of frequently-used commands. These systems support *swapping* (q.v.) instead of *paging* (q.v.) and keep a copy of the text region of sticky bit programs on the swap device. This reduces the setup time on an `exec` for the text pages of a process, because the process is stored in contiguous blocks on the swap device and can thus be loaded with a fast *DMA (Direct Memory Access)* (q.v.) transfer.

On paging virtual memory systems such as SCO UNIX System V/386, swapping has a different meaning, and the sticky bit optimization has less of an impact on system performance. Text pages are often paged right from the file system, and the swapped-out process is not necessarily stored contiguously on the swap device. Moreover, pages of frequently executed programs are often cached in the page cache, a list of pages that are available for reassignment.

The commands and system calls on System V/386 support sticky bits for historical reasons. To set the sticky bit without changing the access permissions for the file, use the `+t` argument to the `chmod(C)` command:

`chmod +t program`

To identify processes that have the sticky bit set, use the following command:

`find / -perm -1000 -print`

Only the superuser can set and unset the sticky bit.

storclass.h header file

The `<storclass.h>` header file specifies constants used to identify the storage class of a COFF (*Common Object File Format*) (q.v.) *symbol table* (q.v.) entry.

See also “Header files” and “syms.h header file” in this *Encyclopedia*.

STREAMS Interface

STREAMS is a software interface that provides a standardized interface for handling communications protocols. The STREAMS software is developed as either a *device driver* (q.v.) or as a communications protocol module.

STREAMS consists of system calls, kernel routines, and resources built into the operating system. STREAMS is described in detail in the AT&T *STREAMS Primer* and *STREAMS Programmer's Guide*, which are listed in the bibliography in this book.

STREAMS, normally associated with a terminal or network, is a full-duplex processing and data transfer path between a process and a driver or, when using STREAMS pipes, between two processes. It is composed of three parts: the STREAM head, zero or more linearly connected modules, and the STREAM end (the device driver or the other process). It is analogous to a *pipe* (q.v.) at the shell level, but data in a communication STREAM can flow in both directions. The STREAM head is connected to the process when the device is opened.

A process that needs to access a STREAMS device driver uses the STREAM head as its interface via a series of system calls. Data passed to the STREAM head through these system calls is converted into a (STREAMS) message form and is then passed from the STREAM head to the driver or to the first intermediate module or, if there is no module, to the driver. Each module processes the message before passing it on to the next module or the driver. Each module provides only one entry point which is a routine that accepts a message. This ensures a consistent interface between the modules and eases the construction and combination of modules that perform standard services. Messages sent by the device driver will also pass through all the modules before reaching the process. Intermediate modules can be added to the STREAM dynamically.

The following system calls are used to access the STREAMS interface:

STREAMS System Calls

Routine	Description
<code>close(fildes)</code>	closes STREAM pointed to by <i>fildes</i>
<code>getmsg(fd,ctlptr,dataptr,flags)</code>	gets message from STREAM pointed to by <i>fd</i> putting control portion in <i>ctlptr</i> and data portion in <i>dataptr</i> according to <i>flags</i>
<code>ioctl(fildes,request,arg)</code>	<i>request</i> and <i>arg</i> are passed to STREAM pointed to by <i>fildes</i> to be interpreted by the device driver
<code>open(path,o flag [, model])</code>	opens STREAM to device at <i>path</i> according to <i>o flag</i>
<code>putmsg(fd,ctlptr,dataptr,flags)</code>	creates message taking control portion from <i>ctlptr</i> and data portion from <i>dataptr</i> and sends it to STREAM pointed to by <i>fd</i> according to <i>flags</i>
<code>poll(fds,nfds,timeout)</code>	check <i>nfds</i> number of the <i>fds</i> STREAMS to see which can send and receive messages or on which events have occurred. Wait <i>timeout</i> msec for an event to occur
<code>read(fildes,buf,nbyte)</code>	read <i>nbyte</i> characters from STREAM pointed to by <i>fildes</i> into <i>buf</i>
<code>write (fildes,buf,nbyte)</code>	write <i>nbyte</i> characters from <i>buf</i> to STREAM pointed to by <i>fildes</i>

For more information, see the manual pages, the “I/O Subsystem” chapter in Maurice Bach, *The Design of the UNIX Operating System*, and the “System V Transport Layer Interface” chapter in W. Richard Stevens, *UNIX Network Programming*. The seminal discussion of the STREAMS interface is D.M. Ritchie’s “A Stream Input-Output System,” published in Volume II of *UNIX System Readings and Applications*.

Streams (Standard I/O)

A data stream is a facility provided by the *Standard I/O* (q.v.) routines to provide buffered file access. A stream is described by a **FILE ***; the **FILE** structure is described in the `<stdio.h>` header file. Conventionally, there are three standard streams: *stdin*, connected to *file descriptor* (q.v.) 0; *stdout*, connected to file descriptor 1; and *stderr*, connected to file descriptor 2.

The routines necessary to use streams are provided in the Standard C Library, *libc.a*. The routines are listed in the “Standard I/O” article. See Brian Kernighan and Dennis Ritchie, *The C Programming Language*, for more information about using these routines.

String manipulation

In the C programming language a string is a sequence of characters terminated by a null character (`\0`). The length of the string does not include the null character *delimiter* (q.v.); however, enough space must be allocated in *memory* (q.v.) for the string and the null character delimiter. Other languages define strings in other ways.

Most of the string functions are defined in the standard C library and declared in the *string.h* header file. (`sscanf()` and `sprintf()` are declared in *<stdio.h>*.) The following string functions let you compare, concatenate, copy, count characters in, read from, and write to strings.

Routines for string manipulation

Routine	Description
<code>strcat(dst,src)</code>	appends characters of one string to end of another
<code>strchr(s,c)</code>	finds the first occurrence of a character <i>c</i> in a string <i>s</i>
<code>strcmp(s1, s2)</code>	compares characters in one string to those of another
<code>strcoll(str1, str2)</code>	collates two strings
<code>strcpy(dst,src)</code>	copies a string
<code>strcspn(s1,s2)</code>	finds first occurrence of a character from given character set in string
<code>strdup(s1)</code>	duplicates string
<code>strlen(s)</code>	returns the number of characters contained in a string
<code>strncat(dst,src,n)</code>	appends characters to the end of a given string
<code>strncoll(str1,str2,n)</code>	collates two strings until the <i>n</i> th character in the first string is reached
<code>strncpy(dst,src,n)</code>	copies characters to a given string
<code>strxfrm(to,from,maxsize,n)</code>	transforms <i>n</i> characters in the <i>from</i> string
<code>strpbrk(s1,s2)</code>	finds the first occurrence of a character from one string in another
<code>strrchr(s,c)</code>	finds last occurrence of a given character in a string
<code>strspn(s1,s2)</code>	finds first substring from a given character set in a string
<code>strtod(str,ptr)</code>	converts strings to double-precision numbers
<code>strtok(s1,s2)</code>	finds the next token in a string
<code>strtol(str,ptr,base)</code>	converts a string to an integer
<code>strxfrm(to,from,maxsize)</code>	transforms the <i>from</i> string to <i>to</i> ; <i>maxsize</i> indicates the maximum value of <i>to</i>

(Continued on next page)

Routines for string manipulation

(Continued)

Routine	Description
sscanf(s,format,argptr)	reads values from a character string and stores values at given memory location according to a format
sprintf(s,format [,arg])	writes values to a given string according to a format

For more information on string functions, see the printf(S), strcoll(S), string(S), scanf(S), strtol(S), and strtod(S) manual pages in the *Programmer's Reference Manual*.

Stringizing

Stringizing is a practice defined in the ANSI C standard and supported on the System V/386 and System V/286 Development Systems that allows a string to be created from the arguments to a C preprocessor macro. Stringizing uses the following operators:

- # transform macro formal parameters into quoted strings
- ## paste two or more strings together

On System V/286, white space is not allowed between the stringizing operator “#” or “##” and the formal parameter on which it operates.

See *Rationale for ANSI Programming Language — C* in the ANSI C specification.

string.h header file

The `<string.h>` header file specifies routines used to manipulate strings.

NOTE Some programs written for non-System V UNIX systems add an “s” to the end of the word “string” in the *string.h* header *filename* (q.v.). This file does not exist on SCO UNIX systems. Make sure you include the `<string.h>` header file rather than the `<strings.h>` header file when you are developing on an SCO UNIX system.

See also “Header files” and “memory.h header file” in this *Encyclopedia* and the manual page for **string(S)**.

stropts.h header file

The `<sys/stropts.h>` header file contains the *STREAMS*(q.v.) user option definitions. It defines the structures and constants for an `ioctl` on a *STREAM* file descriptor. This file is AT&T SVID Issue 2 and X/Open XPG3 compliant. The `<stropts.h>` header file is a phantom header file that contains a `#include <sys/stropts.h>` statement.

Structure

A structure is a C language data type that includes several data elements or variables (called “members” or “fields”). Structures are defined as in the following example:

```
struct mystruct {
    char    member1;
    char    member2;
    int     member3;
    ...
};
```

In this example, `mystruct` is a structure tag that can be used later to reference this type of structure. For example, the following fragment declares `info` to be a `mystruct` structure, and checks that the value of `member2` in `info` is not equal to 0.

```
struct mystruct info;
if (info.member2 !=0) ...
```

For more information about declaring and referencing structures, see Kernighan and Ritchie, *The C Programming Language*.

Packing structures

By default, the `cc`(CP) compiler stores structures as follows:

- Items of type `char` or `unsigned char`, or arrays that contain these data types, are byte-aligned.
- All other types of structure members are aligned on a 4-byte boundary
- Structures themselves are aligned if any member of the structure must be aligned. This alignment is the same as the most restrictive alignment requirement of any of the structure members.

The `-Zp` option to the `cc` command and the `pack` pragma can be used to control how structure data are “packed” into memory. This can be used to conserve space or to conform to existing data structures.

`-Zp[n]`

specifies that all structures in the module will be stored on *n*-byte boundaries. *n* can be 1, 2, or 4.

#pragma pack([n])

specifies packing for an individual structure that is different than the packing specified on the `cc` command line. This directive also determines the alignment of structure members. Use the `pack()` pragma with no arguments to reinstate the packing given on the command line.

The following table shows the interaction of the `-Zp` option with the `pack` pragma:

Syntax	Compiled with -Zp option?	Action
<code>#pragma pack()</code>	yes	reverts to packing specified on the command line for structures that follow
<code>#pragma pack()</code>	no	reverts to default packing for structures that follow
<code>#pragma pack(n)</code>	yes or no	packs the following structure(s) to the specified byte boundary until changed or disabled

Subsystems

The UNIX operating system kernel has three logical subsystems: I/O, process, and file. The following articles offer related information:

- File
- Kernel
- I/O subsystem
- Process
- Process subsystem

Super block

A super block is a special data block that provides control information for a *filesystem* (q.v.). The super block resides in the beginning of logical block 2 of every filesystem. It contains information necessary to mount a filesystem and access its data, information about free space available, the size of the filesystem, and the number of free inodes.

The super block is read into memory when the filesystem is mounted. Periodically, the *kernel* (q.v.) writes the super block to disk if it has been modified.

The super block contains the following status information. To display the size and count information for a particular filesystem, use the `df -t` command.

Sizes

size of the inode list

total number of inodes available in the filesystem.

size of the filesystem

total number of logical blocks in the filesystem, including the super block, inode list, and data blocks. The size of the data block collection can be derived from this number.

Counts of Available Filesystem Space

free inodes

total number of inodes available for allocation

free blocks

total number of storage blocks available for allocation

Memory-Resident Flags

modification ("dirty") flag

This flag is set when the super block in main memory is modified. When `sync` runs, it checks for this flag and, if it is set, updates the super block on disk and clears the flag. If this flag is not set, `sync` does not update the filesystem or the super block.

read-only flag

This flag is set when the filesystem is mounted as a read-only filesystem; if set, any attempt to write to a file in this filesystem fails. This flag is frequently set on filesystems that contain only source code, or when debugging device drivers, to ensure that no writes are accidentally done to files in the filesystem. This flag also can be set when backing up a filesystem while the system is in multiuser state.

Filesystem Description Information

logical block size (type)

indicates the logical block size of the filesystem. UNIX systems have three logical block sizes: 1 indicates a 512 byte logical block size, 2 indicates a 1024 byte logical block size, and 3 indicates a 2048 byte logical block size. XENIX systems have two: 1 indicates a 512 byte logical block size, and 2 indicates a 1024 byte logical block size.

The super block also contains two array fields used for allocating inodes and storage blocks. Each field has a corresponding lock field used to lock the arrays when they are manipulated. See the "Allocating free blocks and inodes" section of the "Filesystems" article.

Super user

The super user is a privilege category that can access all files and resources on the system regardless of the access permissions set for the file or resource. Traditionally, commands documented in the (ADM) section of the manual pages can be executed only by the super user.

A process is recognized as a super user process if its *effective user ID* (EUID) (q.v.) is 0. A user can obtain super user privileges by logging on as root or by executing the su(C) command with no *name* argument.

Swap space

The swap space is a reserved portion of the hard disk in which modified process memory pages are written when the physical memory pages are needed elsewhere. Swap space must be at least as large as the amount of random access memory (RAM) installed.

The following two methods calculate swap space size:

Method 1: let the divvy utility decide for you

If the system is used under typical multiuser operation (no large applications or the Development System), you can use the default swap space recommended during the installation procedure. **divvy**(ADM) calculates the minimum and maximum size based on the size of the hard disk.

Method 2: calculate the swap space yourself

If your system uses major applications (a large relational database, for example), you should allocate the swap space according to the following formula.

1. Multiply the number of users on the machine by the size of the largest process (running program) that normally runs on the machine.
2. Take the amount of memory installed in your machine and add 1Mbyte. Compare this number with the result from step 1. Use whichever number is larger as your swap space size.
3. Adjust the number upward if multiple users are running different applications that are large. Try adding 1 Mbyte per different large application. (Consider the Development System to be a large application.)

For example, the calculations for an eight-user machine with 4 Mbytes of memory with a typical mix of spreadsheet, database, word processor, and graphics package looks like this:

1. 8 users x 1 Mbyte = 8 Mbytes.
2. 4 Mbytes memory installed + 1 Mbyte = 5 Mbytes, which is smaller than the results of step 1.
3. There are 4 users using different large applications. $4 \times 1 \text{ Mbyte} = 4 \text{ Mbytes}$. Add this to the greater amount from steps 1 and 2 to obtain a viable calculation of swap space: $8 \text{ Mbytes} + 4 \text{ Mbytes} = 12 \text{ Mbytes}$. Because the operating system uses a block size of 1 Kbyte, a swap space allocation of 12 Mbytes translates to 12000 blocks.

Swapping

The term “swapping” originally meant moving a whole process back and forth between disk and main memory. The swapping mechanism on a paged system, such as SCO UNIX System V/386, is not the same as a “true” swapping process. On a paged system, part of a process may be in memory and part on disk. This procedure is called “demand paging” because the kernel can ask for only those pages that it needs to execute the process. The primary use of a swap device is supporting *paging* (q.v.). *Swap space* (q.v.) is always used in 4K chunks, matching the processor *page* (q.v.) size. SCO UNIX systems support multiple swap devices.

A process is referred to as “swapped out” when its *u_area* and page table space in memory have been reallocated. The *u_area* is placed on disk, and the page tables are freed and recreated when the process is “swapped in”. When a process is swapped out, its *proc* structure and all of its *region* (q.v.) structures are still in memory. When a process is to be swapped in, space for the page tables and *u_area* is allocated, and the *u_area* is read in, but no actual data is read in. If necessary, data can be faulted in when the virtual address is referenced. When a process is actually scheduled to run, the page directory entries are set up to point to these page tables.

The kernel process that manages swapping processes in and out of memory is referred to as “process 0,” “sched,” or “the swapper.”

See Maurice Bach, *Design of the UNIX Operating System*, for more information about swapping.

Symbol table

A symbol table is a table, maintained by a compiler or other code translation program, that contains information about the symbols found in the *source code* (q.v.). A symbol table in an object file describes information about the names and functions in that *file* (q.v.). The symbol table for a program is generated by the compiler; it is written to the object file for use in linking and debugging.

Symbolic links

A symbolic link is a file that “points” to another file. Symbolic links are a means by which one file can have two or more names. This effectively allows the same file to appear in more than one directory. Any editing that is done, for example, using any of the filenames will still affect only the one file.

There are two ways to create a symbolic link. The first is with the `-s` option to the `ln(C)` command. The second is within a program using the `symlink(S)` function, for example:

```
symlink(dest, to)
```

This creates a file in the filesystem, specified by the *dest* argument, that contains a string which is the path to the file specified as the *to* argument. When the kernel references a symbolic link, it first opens the symbolic link file, reads the pathname of the file that the link points to, and then parses the pathname to reference the actual file.

The symbolic link file is owned by the current effective user ID (EUID) and effective group ID (EGID); and it is mode 0777. The permission bits on a symbolic link, however, are ignored; therefore, any user with search access to the directory in which the symbolic link resides can find out the contents of the symbolic link.

syms.h header file

The `<syms.h>` header file specifies constants, macros and structures used to define a COFF (Common Object File Format) (q.v.) *symbol table* (q.v.) entry.

See also “a.out.h header file”, “Header files” and “storclass.h header file” in this *Encyclopedia*.

Syntax

The syntax of a language is the set of rules that describe the correct ordering of elements in that language. Analysis of the syntactic structure of a sequence of tokens is called *parsing* (q.v.). The set of token sequences (sentences) that are syntactically correct in a language can be described by a list of sentences, a *grammar*, a *regular expression* (q.v.), or a finite automaton.

Natural languages, programming languages, and commands all have a syntactic structure. Command syntax is the order in which command names, options, option *arguments* (q.v.), and operands are put together to form a command line.

System calls

System calls are the means by which a user-level process requests service from the kernel. System calls can be accessed directly from a C language program or indirectly (by calling a C subroutine) from a program written in another language. To the programmer, system calls are indistinguishable from library routine calls.

All system calls take the form of functions, meaning they can return a value. A return value of -1 usually indicates unsuccessful execution. When a call returns -1, the *error handling* (q.v.) external variable called **errno** is assigned a value which describes the problem. Use the **perror(S)** routine to display an error message. All valid error values are defined on the **Intro(S)** manual page; the specific meanings of **errno** values returned by individual system calls are listed on the individual manual pages. See the **Routines(S)** manual page for a list of all system calls.

System V Interface Definition (SVID)

The System V Interface Definition (SVID) is a standard developed by AT&T. It defines, at the C source code level, the interfaces to the kernel services, the contents of standard libraries and the environment provided by the system. The first issue of SVID was published in the spring of 1985. The second issue of SVID, containing three volumes, was published in 1986. The current issue of SVID, issue three, defines System V, Release 4. It is likely that revised issues will appear in the future. The System V Verification Suite (SVVS) verifies that an operating system conforms to SVID.

SVID includes a Base Interface Definition plus a series of extensions. The Base Interface Definition describes the services that all System V operating systems must provide. The extensions to the base are optional but, when a component is present, it must conform to the specified syntax and functionality.

SVID contradicts POSIX in many ways. Because the United States government requires POSIX-conformance rather than SVID-conformance, The Santa Cruz Operation has chosen to favor POSIX and XPG conformance over SVID. Users attempting to port SVID-conformant applications generally succeed by compiling the code with the `-svid` option, thus including available SVID-specific information.

T

Terminal Access Method (TAM) transition library

The TAM transition library is used to allow TAM programs to be compiled and run under *curses* (q.v.). TAM is the screen management package provided on the AT&T UNIX PC. The TAM transition library maps TAM routines and virtual function keys to the corresponding *curses* routines and virtual function keys thus allowing TAM programs to be compiled under *curses* with little or no modification. See the *tam(S)* manual page and the *User Interfaces Guide* for more information about the TAM transition library.

tam.h header file

The *<tam/tam.h>* header file specifies constants, variables and structures necessary to use the TAM transition library in order to compile a *Terminal Access Method* (q.v.) (TAM) program under *curses* (q.v.).

See also “Header files” in this *Encyclopedia* and the *User Interfaces Guide* manual.

Target machine

Compilers and link editors produce executable object files that are intended to be run on a particular computer. In the case of cross compilers, the compilation and link editing are done on one computer, with the intent of creating an object file that can be executed on another computer. The term, “target machine,” refers to the computer on which the object file is destined to run. In the majority of cases, the target machine is the same computer on which the object file is being created.

tar.h header file

The *<tar.h>* header file specifies constants used in the creation of tape archive files.

See also “Header files” in this *Encyclopedia* and the manual pages for *tar(C)* and *tar(F)*.

tbl

tbl is a pre-processor used with the text-formatter *troff* (q.v.) to format tables. The user embeds a table specification, followed by the data, in a **troff** document. The file is processed by **tbl**, which converts the table specification into the equivalent **troff** codes. The resulting file is then processed by **troff**.

tbl supports a variety of tabular formats. There are global parameters to center the table, expand its width to the full line length, enclose the table in a box, and enclose all the data items in boxes. It is possible to lay out individual lines and columns, setting such features as justification, column width, font, and point size. There is also a high degree of control over the use of horizontal and vertical lines around data items. A repeating header can be specified for a multi-page table.

troff and **tbl** are not included in the SCO UNIX System V/386 Development System, but are available from a number of outside vendors or as public domain software for UNIX, XENIX, and DOS systems. For more information about **troff**, refer to O'Reilly and Associates *UNIX Text Processing*. For more information about **tbl** refer to McGilton and McNabb's *Typesetting Tables on the UNIX System*.

tcap.h header file

The *<tcap.h>* header file specifies constants, macros, variables, structures and routines used with the *termcap* (q.v.) terminal description database.

See also "Header file", "curses", "curses.h header file", and "termcap" in this *Encyclopedia* and the *User Interfaces Guide* manual.

TCB (Trusted Computing Base)

TCB (Trusted Computing Base) is a collection of software that maintains parts of the UNIX system kernel and the *trusted facilities* (q.v.) utilities that reference and maintain relevant *security* (q.v.) data.

TCP/IP

TCP/IP is a family of networking protocols that is widely used on UNIX operating systems. TCP/IP is based on the Internet Protocol Suite developed by DARPA (q.v.) in the 1980's and implemented on the BSD UNIX operating system around 1982. It is supported on most System V Release 3 systems. On SCO UNIX operating systems, TCP/IP is implemented using the STREAMS interface.

TCP/IP contains the following protocols (see also the “OSI (Open Systems Interconnection)” article):

- TCP *Transmission Control Protocol*. Provides for reliable, full-duplex communication between user-level processes. TCP is “reliable”, meaning that messages are guaranteed to be delivered correctly and in order unless there is a hardware failure that causes a user-level error. They are not, however, guaranteed to be delivered in a predictable time.
- IP *Internet Protocol*. Provides packet delivery service for TCP, UDP, and ICMP.
- UDP *User Datagram Protocol*. An alternate to TCP for communication between user-level processes. UDP is considered “unreliable”, meaning that messages are not guaranteed to be delivered, so the application must verify delivery if necessary.
- ICMP *Internet Control Message Protocol*. Handles error and control information between gateways and hosts.
- ARP *Address Resolution Protocol*. A protocol used on some networks to map an Internet address to a hardware address.
- RARP *Reverse Address Resolution Protocol*. Used with ARP to map a hardware address to an Internet address.

For more information about developing applications that use the TCP/IP protocols, see W. Richard Stevens, *UNIX Network Programming*.

termcap

termcap is a terminal description database used by the *curses* (q.v.) screen management package and was the predecessor of *terminfo* (q.v.). Like *terminfo*, it is used so that programs do not have to be aware of the individual characteristics of each terminal on which they are to be run. This makes a program independent of any terminal provided that the terminal is described in the database. Unlike *terminfo*, all the terminal descriptions in the *termcap* reside in a single ASCII file (q.v.) called */etc/termcap*, by default, or in a local *termcap* file indicated by the definition of the **TERMCAP** environment variable. To locate a terminal entry the entire file must be read sequentially until the entry is found. Once a match is found for the terminal type, the definitions, known as capabilities, are read into memory and then used by the process, normally through *curses* library routines, to control the terminal.

A terminal entry in the */etc/termcap* file begins with a list of alternate names for the terminal, each separated by a vertical bar or pipe (|) character. A *vt100* terminal, for example, would appear as:

```
d1|vt100|vt-100|pt100|pt-100|dec vt100
```

If the **TERM** environment variable had been set to any member of this list then the *vt100* capabilities would be loaded by the process at run time. The list of capabilities for a terminal entry immediately follows the list of alternate names.

The capabilities for a terminal are each identified by a two letter symbol and are separated from each other by a colon (:). Capabilities can be of type boolean, string or numeric. In the case, for example, of a *vt100* type terminal the following are some of the capabilities that can be accessed:

bs	bool	Terminal can backspace with ^H
cl	str	Clear screen
co	num	Number of columns in a line

These are represented as part of the *vt100* entry in the file */etc/termcap* as:

```
:co#80:cl=50E[;HE[2J:bs:
```

The numeric capability "co" has the value "#80" indicating that there are 80 columns on a *vt100* terminal screen. The "cl" capability takes the "50E[;HE[2J" control character string which, when sent to the terminal, clears the screen. The presence of the "bs" boolean capability indicates that a key which produces "^H" can be used as a backspace key. Further information on termcap and termcap entries can be found on the *termcap*(F) manual page.

terminfo

terminfo is a terminal description database used by the *curses* (q.v.) screen management package and is the successor to *termcap* (q.v.). **terminfo** is a database containing the compiled descriptions of many terminals. Each description specifies the capabilities of a terminal and how it performs various operations, how many lines it has, and how its control characters are interpreted. A *curses*(S) program refers to such a database at run time to obtain information it needs about the terminal being used.

To enable the same program to run on a variety of terminal types, the **TERM** environment variable must be set. **TERM** specifies the **terminfo** file that defines the characteristics of the terminal currently being used.

terminfo files are supported on both SCO UNIX System V/386 operating systems and XENIX operating systems. **termcap** files are similar but have a slightly different format, and are also supported on all SCO systems.

All **terminfo** files are stored under the */usr/lib/terminfo* directory. This directory has subdirectories named *a, b, c, ..., 0, 1, ..., 9*, and so forth. This directory structure organizes the database files based upon the first character in the name. Each terminal type has a name, such as *wy60* for the WYSE 60 terminal and *vt100* for the DEC VT-100; the first letter of that name determines the subdirectory in which the terminal description is stored. For example, the file for the WYSE 60 terminal is */usr/lib/terminfo/w/wy60*; this file is linked to the *wy60* file, which provides an alias name.

terminfo suffixes

Names can be given suffixes that begin with a hyphen to indicate modes for the hardware or user preferences. The following table defines the conventional suffixes.

terminfo suffixes

Suffix	Meaning	Example
-w	wide mode (more than 80 columns)	vt100-w
-nam	without automatic margins	tvi950-nam
-n	number of lines on the screen	aaa-60
-na	no arrow keys (leave them in local mode)	c100-na
-np	number of pages in memory	c100-4p
-rv	reverse video	c100-rv

Choosing a terminfo name

Most terminals you will want to use are already defined in the **terminfo** files provided with the system. If you do not know the **terminfo** name for the terminal in question, look for files named for the model number or vendor initials. To confirm that this is the correct file, read the compiled description file with the **infocmp**(ADM) command. Much of the output from this command will be cryptic, but the full terminal name is displayed in the first line, along with synonymous names.

See also the **tic**(C), **terminfo**(M), **terminfo**(S), and **terminfo**(F) manual pages and the *User Interfaces Guide* for more information.

termio.h and termios.h header files

The `<sys/termio.h>` header file specifies constants and structures used by **termio** *ioctl*'s (q.v.) on terminal file descriptors.

The `<termios.h>` header file specifies files to include for *POSIX* (q.v.) compliant functions for controlling terminal modes.

See also "Header file" in this *Encyclopedia* and the manual page for **termio**(M).

t_errno variable

The **t_errno** variable records error conditions detected by the *TLI* (q.v.) and *XTI* (q.v.) software. Valid values are defined in the `<xti.h>` and `<tiuser.h>` header files, and are documented on the **Intro**(S) manual page.

Time

A number of system and application programs identify the local clock time. Obvious examples are the electronic mail and file management facilities which attach time and date stamps to mail messages, files and directories. For instance, the header attached to **mail** messages contains the following information:

From person Wed Mar 27 14:09 CST 1991

Similarly, date and time stamps are often required for reports associated with user applications and implementations.

Information about a particular locale's time zone is defined in the **TZ** (q.v.) environment variable.

Internally, the system keeps track of time independently of the **TZ** environment variable. The **time(S)** routine returns the number of seconds since 00:00:00 January 1, 1970, UTC (Coordinated Universal Time, commonly known as Greenwich Meridian Time). This value will overflow at 19:14:08 January 19, 2038, on UNIX machines where time is stored in a 32-bit signed integer.

International time conventions

To account for global time zones and daylight savings time schemes, the **TZ** environment variable can be used to configure the system, or one's personal environment, to reflect local time. Although **TZ** has a few shortcomings, in most cases it can be used internationally to conform one's system to a particular locale. When **TZ** is defined, the routines **ctime(S)**, **localtime(S)**, **mktime(S)** and **strftime(S)** will use it in place of the default time zone. See the **TZ(M)** manual page for further details regarding this environment variable.

Accessing time in user programs

The following routines allow you to access the system time in a user-level program. These functions are further described on the **ctime(S)**, **difftime(S)** and **mktime(S)** manual pages:

asctime	converts a tm structure to a 26-character string that identifies the local day of week, month, day of month, hours, minutes, seconds, and year. See the ctime(S) manual page for the complete description of a tm structure.
ctime	converts UNIX operating system time to a 26-character string that identifies the local day of week, month, day of month, hours, minutes, seconds, and year.
difftime	returns the difference between two calendar times.
gmtime	converts UNIX operating system time to UTC time represented in the format of a tm structure.

localtime	converts UNIX time to local time represented in the format of a <code>tm</code> structure.
mktime	converts a <code>tm</code> structure to the proper local calendar time.
strftime	converts a <code>tm</code> structure to a character string with a user specified format.
tzset	sets time conversion information using the TZ environment variable.

time.h header file

The `<time.h>` header file specifies structures and routines used to manipulate times and dates.

See also “Header files” in this *Encyclopedia* and the manual page for `time(3)`.

timezone variable

See “TZ environment variable.”

tinfo.h header file

The `<tinfo.h>` header file specifies constants, structures and routines needed to use the *terminfo* (q.v.) terminal description database with *curses* (q.v.).

See also “Header file” in this *Encyclopedia* and the *User Interfaces Guide* manual.

TLI (Transport Layer Interface)

TLI (Transport Layer Interface) is a communications interface specification that defines network services and how to invoke them. TLI is based on the transport layer definition for the OSI (*Open Systems Interconnection*) (q.v.) model. It is used to develop both peer-to-peer and client/server applications. TLI provides an abstraction layer so that programmers do not have to interact directly with the underlying protocols. The X/Open Transport Interface version of TLI is discussed in the “XTI” article in this *Encyclopedia*.

TLI provides two modes of service:

connection-mode	circuit-oriented service that enables data to be transmitted over an established connection in a reliable, sequenced manner. It also provides an identification mechanism that avoids the overhead of address resolution and transmission during the data transfer phase.
-----------------	---

connectionless-mode message-oriented service that supports data transfer in self-contained units with no logical relationship required among multiple units. This service requires only a preexisting association between the users involved. This determines the characteristics of the data to be transmitted. All the information required to deliver a unit of data is presented to the transport provider, together with the data to be transmitted, in one service access (which need not relate to any other service access). Each unit of data transmitted is entirely self-contained.

TLI routines

The Network Services Library of the UNIX System include the following set of TLI routines. The XTI library contains the same routines. Specify the **-l libnsl** or **-l libnsl_s** option to the CM cc CP command to access the TLI routines; use the **-l libxti** or **-l libxti_s** option to access the XTI routines. See the “XTI” article for information about converting TLI applications to XTI.

TLI routines for either service mode

Function	Description
t_alloc(S)	allocates TLI data structures
t_bind(S)	binds a transport address to a transport endpoint
t_close(S)	closes a transport endpoint
t_error(S)	prints a TLI error message
t_free(S)	frees structures allocated using t_alloc
t_getinfo(S)	returns a set of parameters associated with a particular transport provider
t_getstate(S)	returns the state of a transport endpoint
t_look(S)	returns the current event on a transport endpoint
t_open(S)	establishes a transport endpoint connected to a chosen transport provider
t_optmgmt(S)	negotiates protocol-specific options with the transport provider.
t_sync(S)	synchronizes a transport endpoint with the transport provider.
t_unbind(S)	unbinds a transport address from a transport endpoint

Connectionless-mode routines

Function	Description
t_rcvudata(S)	retrieves a message sent by another transport user
t_rcvuderr(S)	retrieves error information associated with a previously sent message.
t_sndudata(S)	Sends a message to the specified destination user

Connection-mode routines

Function	Description
<code>t_accept(S)</code>	accepts a request for a transport connection
<code>t_connect(S)</code>	establishes a connection with the transport user at a specified destination
<code>t_listen(S)</code>	retrieves an indication of a connect request from another transport user
<code>t_rcv(S)</code>	retrieves data that has arrived over a transport connection
<code>t_rcvconnect(S)</code>	completes connection establishment if <code>t_connect</code> was called in asynchronous mode
<code>t_rcvdis(S)</code>	returns an indication of an aborted connection, including a reason code and user data
<code>t_rcvrel(S)</code>	returns an indication that the remote user has requested an orderly release of a connection
<code>t_snd(S)</code>	sends data over an established transport connection
<code>t_snddis(S)</code>	aborts a connection or rejects a connect request
<code>t_sndrel(S)</code>	requests the orderly release of a connection

For more information, refer to the manual pages, the "Transport Layer Interface" chapter in *Developer's Topics* and the "System V Transport Layer Interface" chapter in W. Richard Stevens, *UNIX Network Programming*.

Translation Lookaside Buffers (TLB)

The 386 central processing unit contains a Translation Lookaside Buffer (TLB), which is a cache for translating page table linear addresses into physical addresses. This cache speeds up memory access by avoiding the actual fetch of the page table entry for recently accessed pages. The kernel flushes the TLB when its entries are out of date.

troff

Properly speaking, **troff** is a text formatting system developed at AT&T Bell Laboratories during the 1970s to produce output for the C/A/T phototypesetter. Now, however, the term **troff** is commonly used to refer to the multitude of descendants, both commercial and public-domain, of the original **troff**. Many versions of **troff** are device-independent — instead of producing formatting codes for a specific device, they generate an intermediate language that post-processors convert to device-specific codes (such as PostScript), that can be printed on a variety of laser printers as well as phototypesetters.

troff documents contain text combined with typesetting instructions, and are created with an ordinary text editor. The documents, unlike those produced by so-called WYSIWYG¹ text formatters, do not take on their final, formatted appearance until they have been processed by **troff** and sent to an output device. The disadvantage of this is that the user does not get to see immediately

what the document will look like. The advantages of **troff** are that it is less resource-intensive than screen-oriented formatters, gives the user a great deal of control over page layout, and is especially well-suited to processing large, structured documents.

troff is not included in the SCO UNIX System V/386 Development System, but is available from a number of outside vendors or as public domain software for UNIX, XENIX, and DOS systems. For more information about **troff**, refer to O'Reilly and Associates *UNIX Text Processing*.

Trusted facilities

The trusted facilities provide level C2 security as defined by the United States Department of Defense "Orange Book." See the "Security" article for an explanation of security levels.

The *User's Guide* and *System Administrator's Guide* provide information about using the trusted facilities built into the operating system to protect the kernel and other standard system resources. The information here provides background for developing trusted applications, in other words, applications that use the trusted facilities to protect the integrity of their data.

Trusted facilities concepts

Two hierarchies record trusted privileges:

/tcb/files/auth/[a-z] files that specify privileges for each user. The subdirectories correspond to the first initial of the UID (so user doug is in the *d* directory and user larry is in the *l* directory).

/etc/auth files that specify privileges associated with programs and applications. The *system* subdirectory defines global system settings; the *subsystem* subdirectory defines the authorizations associated with each subsystem.

The format of these files is described on the *authcap*(F) manual page.

Each trusted application uses the routines defined for the trusted facility to record potential security breaches in the audit database.

Implementing a trusted application

To implement a trusted application:

1. Create a subsystem file for the application in the */etc/auth/subsystems* directory. For instance, for the **myapplic** application, create the */etc/auth/subsystems/myapplic* file. This file is populated just like other security files; see the *User's Guide* and *System Administrator's Guide* for instructions.

2. Determine the privilege granularity needed for the application and design the application to break up functionality accordingly. For instance, if some users are authorized to read the data but not write the data, put the read and write code in separate programs.
3. Issue the `set_auth_parameters` call as the first operation in the `main()` routine in each program in the application.
4. Write the code using the routines in the following sections to check privileges before doing an operation and to record potential security breaches in the audit database.

Routines for implementing a trusted application

The following tables summarize the routines used to implement a trusted application. Because these routines are in the *prot* library, code that uses them must be compiled with the `-lprot` option to `-cc`.

The routines on the `authaudit(S)` manual page record possible security breaches. Note that these routines merely record the information; they do not identify potential security breaches.

Recording possible security breaches (authaudit(S))

Routine	Description
<code>audit_security_failure(obj, expct, curr, act, res)</code>	records a problem involving the specified <i>object</i>
<code>audit_subsystem(subsystem_type, action, result)</code>	records significant event in a specific subsystem
<code>audit_authentry(desired_entry, type, problem)</code>	reports a database inconsistency
<code>audit_no_resource(resource, object, problem)</code>	records that system resource was unavailable; records that the unavailability of a system resource caused a security operation to be aborted
<code>audit_lax_file(path, problem)</code>	reports a breakdown of a sanity check on the proper setup of system files

The routines listed in the following table retrieve the user and group identifications that were in effect when the program began execution. They may return different ID's than `getuid(S)`, `getuid(S)`, `geteuid(S)`, and `getgid(S)` because intervening calls to `setuid(S)`, `setuid(S)` or `setgid(S)` can change the results of those routines, whereas the results of the routines listed below are not affected.

Get user and group id's from start of program

Routine	Description
<code>set_auth_parameters(argc, argv)</code>	retains these ID's for later use. This call must be the first call in the <code>main()</code> routine of any trusted program
<code>is_starting_luid(uid)</code>	compares <i>uid</i> to login user UID
<code>is_starting_ruid(uid)</code>	compares <i>uid</i> to real user UID
<code>is_starting_euid(uid)</code>	compares <i>uid</i> to effective user UID
<code>is_starting_rgid(uid)</code>	compares <i>gid</i> to real group GID
<code>is_starting_egid(uid)</code>	compares <i>gid</i> to effective group GID
<code>starting_luid()</code>	returns login UID for process
<code>starting_ruid()</code>	returns real UID for process
<code>starting_euid()</code>	returns effective UID for process
<code>starting_rgid()</code>	returns real GID for process
<code>starting_egid()</code>	returns effective GID for process
<code>check_auth_parameters()</code>	verify that <code>set_auth_parameters</code> has been run

Routines used to implement trusted facilities commands

Access audit section data (audit(S))

Routine	Description
<code>audit_open(session)</code>	opens database of possible security breaches records
<code>audit_read</code>	retrieves audit records sequentially from audit session data file
<code>audit_close</code>	terminates processing for session

Get information from authentication database (authcap(S))

Routine	Description
<code>agetuser(user)</code>	finds entry associated with specified user
<code>agetcommand(filepos, fd)</code>	finds next command entry in field described by <i>fd</i> starting at the <i>filepos</i> position within the file
<code>agetfile(filepos, fd)</code>	finds next file entry in the file
<code>agetty(filepos, fd)</code>	finds next tty entry in the file
<code>agetdefault(filepos, fd)</code>	returns number associated with <i>id</i> argument
<code>agetstr(id, area)</code>	returns string associated with <i>id</i> argument
<code>agetflag(id)</code>	returns the flag associated with <i>id</i> argument
<code>asetdefaults(new_defaults)</code>	sets location of system values

tty structure

The `tty` structure is the central point of a *line discipline* (q.v.). It is used by a line discipline to point to I/O *buffers* (q.v.) and to store flags, states, and conditions used when handling data from a tty device. `<sys/tty.h>` defines this structure. An instance of this structure is allocated for every tty device including terminals, pseudo-ttys, and for each `mcscreen`; that is, every `/dev/tty*` file has an associated `tty` structure instance.

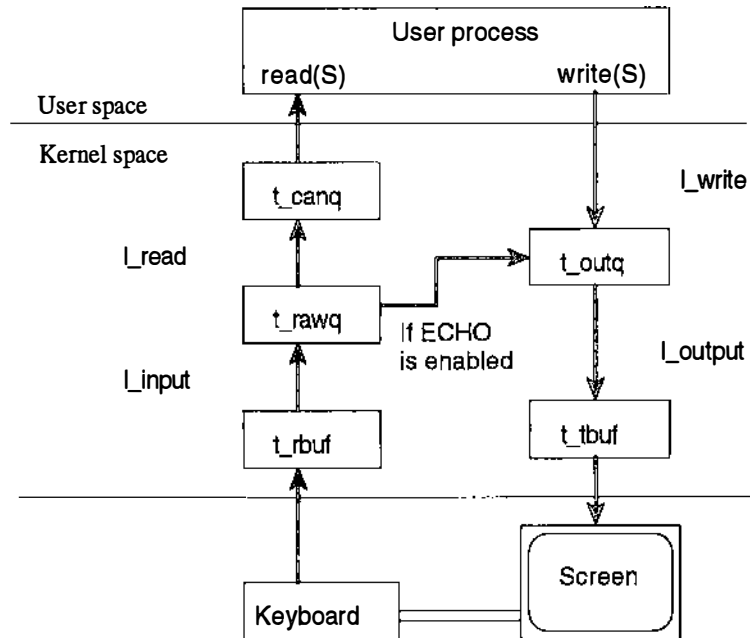
Refer to `tty.h` for descriptions of the `cblock`, `ccblock`, and `clist` structures used in the `tty` structure's definition.

The fields in the `tty` structure are:

Type	Field	Description
clist	<code>t_rawq;</code>	Raw input queue
struct clist	<code>t_canq;</code>	Canonical queue
struct clist	<code>t_outq;</code>	Output queue
struct ccblock	<code>t_tbuf;</code>	Transmit control block
struct ccblock	<code>t_rbuf;</code>	Receive control block
int	<code>(*t_proc) ();</code>	Routine for device functions
ushort	<code>t_iflag;</code>	Input modes
ushort	<code>t_oflag;</code>	Output modes
ushort	<code>t_lflag;</code>	Line discipline modes
ushort	<code>t_state;</code>	Internal state
short	<code>t_pgrp;</code>	Process group name
short	<code>t_line;</code>	Line discipline index
char	<code>t_delct;</code>	Delimiter count
char	<code>t_term;</code>	Terminal type
char	<code>t_tmflag;</code>	Terminal flags
char	<code>t_col;</code>	Current column
char	<code>t_row;</code>	Current row
char	<code>t_vrow;</code>	Variable row
char	<code>t_lrow;</code>	Last physical row
char	<code>t_hqcnt;</code>	Number of high queue
		Packets on <code>t_outq</code>
char	<code>t_dstat;</code>	Used by terminal handlers and line disciplines
unsigned char	<code>t_cc[NCC+5]</code>	Setable control chars
unsigned char	<code>t_schar;</code>	Save timeout characters instead of using <code>t_lflag</code>

For more information on `tty` structure fields, refer to the *Device Driver Writer's Guide*.

The following illustration shows how the fields in the `tty` structure are used when data is conveyed between a user process and a terminal device:



Data received from the keyboard is stored in the `t_rbuf` receive buffer by the `tty` device driver. The *line discipline* (q.v.) `l_input` kernel routine, generally `ttin(K)`, conveys the data into the `t_rawq` raw data queue. If ECHO is set in `c_lflag`, then data is echoed as it is received.

Then *canonical processing* (q.v.) occurs translating the received characters as they are moved from `t_rawq` to `t_canq`. This processing is performed by the `l_read` kernel routine, generally `canon(K)`. From the `t_canq` buffer, the *kernel* (q.v.) moves the data to the user level buffer allocated by `read(S)`.

When data is written by a user level program, the kernel moves into the `t_outq` buffer using the `l_write` routine, generally `ttwrite(K)`. Then data is moved from `t_outq` to the `t_tbuf` transmit buffer using the `l_output` routine, generally `ttout(K)`. From the transmit buffer, data is then moved to the terminal display by the device.

Assigning values to structure members

The `tty` structure can be assigned values using the user level `stty(C)` utility and the `ioctl(S)` system call. Refer to the `termio(M)` manual page for information about the possible values that can be set in the `tty` structure fields. The `stty` utility is used to change options for the fields. To change the values in the structure itself, use the `ioctl` system call. Refer to the `termio(M)` manual page for a list of `ioctl` commands for examining, setting values, and writing information to the `tty` structure.

Type size summary

The following table summarizes the data type sizes as defined in the Microsoft C Compiler, Version 6.0 for the 386 microprocessor.

C Type sizes for Microsoft 6.0 compiler

Type name	Alternate names	Storage	Range of values
<code>char</code>	signed char	1 byte	-128 to 127
<code>int</code>	signed signed int	4 bytes	-2,147,483,648 to 2,147,483,647
<code>short</code>	short int signed short signed short int	2 bytes	-32,768 to 32,767
<code>long</code>	long int signed long signed long int	4 bytes	-2,147,483,648 to 2,147,483,647
<code>unsigned char</code>	—	1 byte	0 to 255
<code>unsigned</code>	unsigned int	4 bytes	0 to 4,294,967,295
<code>unsigned short</code>	unsigned short int	2 bytes	0 to 4,294,967,295
<code>unsigned long</code>	unsigned long int	4 bytes	0 to 4,294,967,295
<code>enum</code>		4 bytes	0 to 4,294,967,295
<code>float</code>		4 bytes	7-digit precision (approx. 3.4E-38 to 3.4E+38)
<code>double</code>		8 bytes	15-digit precision (approx. 1.7E-308 to 1.7E+308)
<code>long double</code>		8 bytes	15-digit precision (approx. 1.7E-308 to 1.7E+308)

NOTE Any type size modified by the `unsigned` keyword can be modified by the `signed` keyword instead. The `signed` keyword is useful if the `-J` option has been used to change the default sign of the `char` type.

types.h header file

The `<sys/types.h>` header file specifies constants, macros, structures and types used within *kernel* (q.v.) code and other header files in the *sys* subdirectory.

See also “Header file” in this *Encyclopedia*.

TZ environment variable

The TZ environment variable contains information about the local time zone. TZ is defined in the */etc/rc*, */etc/default/login* and */etc/profile* files. In the special case where a user is located in a different time zone than that of the host machine, TZ may be set in the user’s *.profile* or *.login* file in their home directory. When TZ is defined, the `ctime(S)`, `localtime(S)`, `strftime(S)` and `mktime(S)` routines use the timezone it defines in place of the default time zone.

UNIX System V convention

As mandated by the POSIX standard, the value of TZ can be set using one of two formats: (spaces inserted for readability)

std offset dst offset; rule

or

:characters

The expanded format of TZ, using the first method is:

sss[ddd[m][;start[/time],end[/time]]]

The meanings of and acceptable values for each of these fields are:

sss Three to nine letters designating the standard time zone. Both upper and lowercase letters are permitted.

n Number of hours past Greenwich Mean Time for the standard time. *n* can be a single integer designating the number of hours or it can be broken up into partial hours using the following format:

hh[:mm[:ss]]

Positive values represent locales west of Greenwich, while negative values are east of Greenwich. In the latter case, *n* begins with a leading minus (-) whereas in the former case, a leading plus sign (+) is optional.

ddd Three to nine letters designating the local daylight savings time (summer time) zone. If not present, summer time is assumed not to apply.

m Number of hours past Greenwich mean time for the summer time. The time is represented in the same format as the one used for the field *n*. If *m* is not given, the distance to GMT during summer time is assumed to be one hour less than during standard time.

start The rule defining the day that summer time begins. In the southern hemisphere, the ending day is earlier in the year than the starting day. If the *start* and *end* fields are not included, the default dates are the days defined by current U.S. law. The manner in which a day is specified can be one of the following:

Jn 1 based Julian day n (1 - 365)
 n 0 based Julian day n (0 - 364)
 Wn.d day d (0 - 6) of week n (1 - 53)
 Mm.n.d day d of week n (1 - 5) of month m (1 - 12)

See the **timezone(F)** manual page for further explanations and exceptions to this scheme.

end The rule defining the day summer time ends.

time The time of day the change to and from summer time occurs. The default is 02:00:00 local time.

If TZ is defined according to the second format (a string with a leading colon), the characters following the colon are interpreted according to the specific definition of a particular application or implementation. Such schemes as double daylight time, solar time, or multiple daylight times cannot be represented using the first format. Therefore, this second format is included because of these limitations imposed by the first format. The second format allows for application specific time interpretation such as a string-based lookup table.

Setting the TZ variable

To set TZ to the Eastern Standard/Daylight Time, enter the one of the following, depending on which shell is being used:

For *Korn Shell (ksh)* (q.v.) or *Bourne shell (sh)* (q.v.):

\$TZ=EST5EDT; export TZ

For *C shell (csh)* (q.v.):

setenv TZ EST5EDT

The fully expressed TZ string for Eastern Standard/Daylight Time, using the current U.S. law of changing to daylight savings time on the first Sunday in April, and back to standard time on the last Sunday in October at 2:00 a.m. local time, is:

EST05:00:00EDT04:00:00;M4.1.0/02:00:00,M10.5.0/02:00:00

Localization

On most systems, by setting the TZ, LANG (q.v.) and the various *locale* (q.v.) environment variables, one can completely localize the system so that language, dates, times and other related data conform to local conventions.

U

UART (Universal Asynchronous Receiver/Transmitter)

The UART is a hardware device used for communicating over a serial line, such as the line to a terminal or a modem. It has two data handling modes, receiving and transmitting. When receiving, the UART collects bits from a serial data line and holds them until enough are received to construct a *byte* (q.v.). The byte is then sent to the processor for handling. In transmission mode, a byte is broken down and the bits are sent out on a serial line one-by-one.

In addition to the data lines, the UART provides control signals to determine when a character has been received, when the UART is ready to send another character, and other functions. One or more control registers are usually provided to allow the setting of such options as the number of bits in each character, the number of stop bits, and the type of parity bit to be generated.

unistd.h header file

The `<sys/unistd.h>` header file specifies constants used by the *POSIX* (q.v.) specification.

See also "Header file" in this *Encyclopedia*.

UNIX Operating System

The UNIX Operating System, developed by AT&T, is a general-purpose, multi-user, interactive time-sharing operating system that uses a modular, hierarchical design structure to permit complex *programs* (q.v.) to be built from simpler ones. The UNIX Operating System also provides a simple consistent interface to *files* (q.v.) and peripheral devices to ease the writing of *application programs* (q.v.). It also "hides" the machine architecture to make it easy to write programs that run on any hardware platform for which the UNIX Operating System has been implemented.

The UNIX Operating System is divided into two parts. The first part is called the *kernel* (q.v.), which is the operating system itself. The kernel is the layer that wraps around the machine architecture and hides its idiosyncrasies. It provides a consistent interface through which programs can make use of system services. This allows programs to be written that are independent of the

underlying hardware and that can be easily moved from one UNIX installation to another. The second part is the *utilities* (q.v.), programs that perform specific tasks. These programs use *system calls* (q.v.) to request services from the kernel. These system calls provide access to the hardware resources and allow the transfer of data between program and the kernel. Most utilities perform some small, specific task but these can be combined with other utilities and user written programs to perform more complex tasks. User written programs can be added to perform additional tasks, as required; these programs also request services from the kernel.

Interaction between the UNIX Operating System and the user is provided through the *shell* (q.v.). This interaction allows the user to execute *commands* (q.v.) by typing their names on the command line.

The difference between files and peripheral devices is also “hidden” by the UNIX Operating System. This allows all data transfer between programs and both files and devices to be done through a common interface. The file or device is referenced through use of a *file descriptor* (q.v.). The file descriptor is used to read and write to a file or device, which simplifies the writing of application programs because it eliminates the need to use different I/O methods for different devices.

User address space

User address space is the portion of the virtual memory address space set aside for the execution of user *processes* (q.v.). The 80386/486 hardware allows virtual memory to be as large as four gigabytes. Only part of this, however, is available for user programs, as determined by the MAXMEM tunable parameter. MAXMEM has a default value of 2560 pages of 4 kilobytes each or 10 megabytes. This parameter can be altered using the *configure(ADM)* utility.

NOTE The size of the virtual address space allowed for programs is independent of the amount of physical memory on the system. MAXMEM can often be larger than the amount of physical memory.

User ID (UID)

The user ID (UID) is the unique number (a positive integer) assigned to each user in the *passwd file* (q.v.), and used to control access to system files and commands. Programs can access this value by looking at the *pw_uid* member of the *passwd* structure that is declared in the *<pwd.h>* header file.

Each process has an associated UID that determines the process’s privileges. Usually the process’s UID is the same as that of the user who issued the command that created the process. Under some circumstances, a process can execute with a different UID than its invoker; this happens if the program has the *setuid* bit set, or if the process executes the *setuid(S)* system call.

A UID is also associated with each file in a file system. This determines the owner of a file. If the UID of a process matches the UID of a file, the allowed access is determined by the “owner” portion of the files permission field.

See also the “EUID (Effective-user ID),” “RUID (Real-user ID),” and “Permissions, file access” articles in this *Encyclopedia*.

user structure

When a *process* (q.v.) is created, an instance of the `user` structure is allocated which contains operating system information for that process. Additional information is contained in the process table. The `user` structure, also called the “u_area” or the “ublock”, is part of the kernel address space and is not directly accessible to an executing program. If a process is swapped out, its “u_area” may also be swapped out.

The `user` structure is defined in the `<sys/user.h>` header file (q.v.). Refer to the *Device Driver Writer's Guide* for more information.

/usr/group

“/usr/group” is an organization formed in February 1983 to attempt to define a standard UNIX operating system. Their Uniform 1983 Draft Standard and Uniform 1984 Draft Standard were the first UNIX standards prepared by representatives of several different companies and institutions. These standards were a direct predecessor of the IEEE effort that eventually produced the POSIX (q.v.) standard.

ustat.h header file

The `<sys/ustat.h>` header file specifies a structure used when getting file system statistics.

See also “Header file” in this *Encyclopedia* and the manual page for `ustat(5)`.

Utility

A utility is a program, usually provided by the operating system, to perform common, computer-oriented tasks. Utility programs are best understood in contrast to *application* (q.v.) programs which perform high-level, project-oriented tasks. Following this distinction, a copy or sort program would be considered a utility, while an accounting program or word processor would be considered an application.

The UNIX operating system provides a full complement of user, programmer and administrator *commands* (q.v.) that function as utilities. Additionally, utilities can also include shell scripts and *high-level language* (q.v.) programs written to perform specific tasks.

utime.h header file

The `<utime.h>` header file specifies a structure used to change the access and modification time on a file.

See also “Header file” in this *Encyclopedia* and the manual page for **utime(S)**.

utmp.h header file

The `<utmp.h>` header file specifies constants and structures that describe the record format used in the `/etc/utmp` and `/etc/wtmp` files.

See also “Header file” in this *Encyclopedia* and the manual page for **utmp(F)** and **wtmp(F)**.

V

values.h header file

The <*values.h*> header file specifies constants and macros that define commonly used floating point values and the limits on values that can be stored in C data types.

See also “Header files” in this *Encyclopedia*.

varargs.h header file

The <*varargs.h*> header file specifies constants, macros and routines used by C programs for functions that have a variable number of *arguments* (q.v.).

See also “Header files” in this *Encyclopedia*.

Variable

A variable is a location in memory used as mutable storage for data. Most often you define and reference variables with symbolic names that are chosen to reflect the intended use of the variable. There are three types of variables of interest on UNIX systems: program variables, shell variables and environment variables.

Program variable

Program variables are used in *programs* (q.v.) for values that are evaluated during the execution of the program. You can use them to store computations, results, counter values, and so on. In most programming languages you must define a variable before using it. This allocates space in memory that can later be referenced through the variable name. In strongly typed languages such as C, program variables are defined to be of a particular data type, for example *int* or *float*.

For information about variables in a specific language, refer to reference material for that language. See the “Bibliography” in this *Encyclopedia*.

Shell variable

Shell variables are used in the shell much as program variables are used in programs. Shell variables are assigned values in the *Bourne shell*, *sh* (q.v.), and the *Korn shell*, *ksh* (q.v.), by typing the name of the variable, an equals sign (=) and a value, for example:

```
$ message='This is a test message'
```

This assigns the string **This is a test message** to the shell variable **message**. Note that there can be no *white space* (q.v.) between the variable name, the equals sign and the value. The value can contain white space as long as it is enclosed in single quotes as in the above example.

Shell variables are assigned values in the C shell with the **set** command. For the correct usage of **set** see the *csh(C) manual page* (q.v.).

The value of a shell variable can be referenced by preceding the variable name with a dollar sign (\$). The **echo** command can be used to display the value of a shell variable, for example:

```
$ echo $message
This is a test message
```

Shell variables are all of the string data type; numbers are treated as characters and stored as the appropriate string. When used in shell scripts, shell variables do not need to be defined as do program variables; they are simply assigned a value when they need to be used.

The output from UNIX commands can be assigned to shell variable by enclosing the commands in back quotes (`), for example:

```
$ count=`ls | wc -l`
```

This assigns to the variable **count** the output of the command, namely the number of *files* (q.v.) that are in the current *directory* (q.v.). Shell variables can be compared and tested in shell scripts with the **test(C)** command.

Some shell variables have special meaning and are set automatically by the shell when a command is executed. The variables **\$0**, **\$1**, ..., **\$9** represent the first ten command *arguments* (q.v.). **\$#** is the number of arguments (excluding **\$0**). The **\$*** and **\$@** shell variables are set to a value of all the arguments collectively. Except when quoted by double quotes ("), **\$*** and **\$@** are equivalent. However, when **\$*** is enclosed in double quotes, it is equivalent to a single string comprised of all the arguments separated by spaces. When **\$@** is double quoted, it is equivalent to individually quoting each argument (separated by spaces).

The shell does not permit directly assigning values to the number variables or to the `$*` variable. However, the `set` command can be used to assign values to these variables. The script that follows shows how `set` can be used to capture the output from another command and display it:

```
PATH=/bin:/usr/bin
set --`who am i`
echo I am $1 and I logged on at $5
```

In the example, the `set` command assigns the output of the `who am i` utility to variables `$0` through `$5`, then the next line displays the login name and the login time.

Environment variables

Environment variables are similar to shell variables except that they can be accessed by shells and programs other than the one in which they were defined. Environment variables are often used by commands to determine where they should look for particular files or other resources.

Environment variables are created through the Bourne shell and the Korn shell with the `export` command. For example:

```
$ message='This is a test message'
$ export message
```

In the C shell, environment variables are set and exported with a single command called `setenv`. Refer to the `csh(C)` manual page for more information about `setenv`.

Once the variable has been exported it becomes available to shells that are started as subshells, either interactive shells that are started from the command line or shells that are started by executing shell scripts, as well as to programs started from these shells.

Some environment variables are set automatically during login. Depending on the shell, these may include `HOME`, which is a path to the user's home directory, and `PS1` and `PS2`, which set the command prompt strings. Environment variables that are currently set can be displayed with the `env(C)` command or the `set` and `export` commands depending on the shell being used. Check the man page for the shell you are using.

vhand

vhand, the “page out daemon”, is a UNIX *kernel* (q.v.) *process* (q.v.) that frees up memory. It is “woken up” at intervals specifically to look for memory to free. The behavior of **vhand** is controlled by a set of tunable parameters that are described in the *SCO UNIX System V/386 Operating System, System Administrator's Guide*. By default, **vhand** looks for pages to steal when less than GPGSLO pages of memory are free. When this occurs, **vhand** pages out the least recently used pages on the system (that is, it moves them to the free list and arranges to write their contents to the disk) until GPGSHI pages of memory are free.

vhand does a pass over of all unlocked *regions* (q.v.) in the system. During the first pass, only regions that have been aged are looked at. If this does not free up sufficient memory, **vhand** does another pass through all regions. During this pass, pages can be stolen from all regions.

In 386 SCO UNIX systems, the MAXSC and MAXFC tuneables (the number of *pages* (q.v.) to accumulate before doing the actual *swap* (q.v.) or free) are always set to one. This means that once a page is determined to be swappable, it is immediately swapped.

See also “paging” and “swapping” in *Encyclopedia*.

Virtual address mapping

The UNIX System is a virtual memory operating system. This means that all addresses used by a running *process* (q.v.) are virtual addresses. Virtual addresses are mapped by the hardware into specific physical addresses. The hardware's memory management unit (MMU) translates the virtual address into a physical *page* (q.v.) frame number.

The details of the mapping are complex and hardware specific. Typically, a virtual address is divided into a virtual page number and an offset in the page. The page number is mapped via the page tables into a physical page address. This is then concatenated with the byte offset to form a complete physical address.

The key to performing the virtual-to-physical address translation, or mapping, is the page map maintained in *kernel* (q.v.) memory and defined in the *immu.h* header file. When a process is running, its page table is loaded into the page map.

W

White space

White space is a string of one or more of the following characters: space, horizontal tab, new-line, vertical tab, and form-feed. It is used by many programs and programming languages to separate strings of characters that represent distinct entities. For instance, it is used by the shell on the command line to separate a *command* (q.v.) from an *argument* (q.v.) and to separate any additional arguments.

Widget

Widgets are user interface components of *X Window System* (q.v.) client applications. X Window System users perceive widgets as buttons, menus, scrolling text or drawing areas, or any other on-screen objects, that exhibit consistent behavior from one application to the next. In general, client applications are developed with a single set of widgets to ensure consistent look and feel throughout. For example, client applications developed for SCO Open Desktop use the OSF Motif widget set, known for its distinctive chiseled appearance.

To help application programmers create consistent user interfaces, widget sets are supplied in the form of development system libraries. The X Intrinsics provide a set of basic widgets, but application developers usually turn to additional widget sets to create client applications.

The X Intrinsics include standard procedures for including widgets in client applications, but widget sets may also include procedures for including specific types of widgets in client applications. For example, to add a Motif push-button widget to a client, you could use either the `XiCreateWidget` Intrinsics procedure or the `XmCreatePushButtonWidget` Motif procedure.

Users can customize the appearance and behavior of individual clients by modifying resources associated with the client's widgets; the user only needs the widget and resource names. Ultimately, the degree to which a user can customize a client is established by the client application programmer.

For more details on programming with the OSF/Motif widget set, see Douglas Young's *The X Window System: Programming and Applications* book.

Window (*curses*)

A window refers to a rectangular area located on any part of a terminal screen that can be controlled independently of any other portion of the screen. A *curses* (q.v.) window is not a physical object but rather a data *structure* (q.v.) that contains an internal representation of a window. All the *curses* (q.v.) I/O routines modify one of these internal representations rather than having any direct effect on the terminal screen. When the *curses* routine *wrefresh* is called the screen is updated by mapping the internal representation to the corresponding physical location on the screen.

Two windows are provided by default to any program that uses the *curses* package. The first is *stdscr*, the standard screen, which represents the entire terminal screen and is the window directly affected by all the *curses* routines that do not specify a particular window. The second is *curscr*, the current screen, which represents what is currently displayed on the terminal. When the *wrefresh* routine is called *stdscr* is compared with *curscr* and changes that are found in *stdscr* are updated on the physical terminal screen. *curscr* is then modified to be identical to *stdscr*.

curses also provides the ability to create other windows in addition to the default *stdscr*. They can be placed anywhere and made any size that remains within the dimensions of the physical screen, and can overlap other windows as well. A separate set of *curses* routines exist to interact with these additional windows. When the *wrefresh* routine is called on one of these additional windows it is compared with all other windows to determine how much of the window is visible and then with *curscr* to determine any changes. These changes are then updated on the terminal screen. For additional information on *curses* window routines see the *curses*(S) manual page, "Curses" in this *Encyclopedia*, or the *User Interfaces Guide*.

Word

A word is a unit of computer *memory* (q.v.) which consists of a number of *bytes* (q.v.) of information. The number of bytes in a word depends on the architecture of the computer. The 80286 has 2 bytes (16 bits) per word; the 80386 has 4 bytes (32 bits) per word.

Although computers can reasonably be thought of as operating on bytes for performance reasons the hardware itself is often structured in terms of words. For example, memory reads and writes are done one word at a time, with additional circuitry provided to merge existing parts of a word if less than a full word is written. Because of this organization, some operations can only be performed efficiently on an appropriate word boundary. This affects the addresses that can be used to store integers or pointers, for example. This, in turn, determines the packing provisions for *structures* (q.v.).

X

X Window System

The X Window System is a device independent window system based on an architecture in which an application *process* (q.v.), called a client, communicates via a network protocol with the server, a process that controls the keyboard, screen, and mouse. The X Window System differs from most window systems in that the application program can be physically separated from the display where the output of the program appears. This is accomplished by means of a protocol in which a client utilizes a predefined set of commands to communicate with the server.

By separating the client application program from the server that controls the display and inputs, X achieves network transparency. In addition, the input, drawing and interclient synchronization code exists only in the server, as opposed to within each client.

Most X clients are built with user interface components called *widgets* (q.v.). Application programmers obtain widgets in the form of data structures that define how user interface components appear and a set of associated procedures that determine what happens when the widget is activated (clicked, double-clicked, and so on) by the user. This model closely resembles the "object and methods" model used in object-oriented programming. As a result, object oriented programming is particularly well-suited for programming clients for X.

Programming libraries exist for many operating systems to allow applications to operate as X clients. Xlib is the C language binding to the X Protocol, though it is rarely used on its own by X client developers; several other high-level toolkits, such as OSF/Motif, exist that provide the widgets that give X clients distinctive and consistent "look and feel." Libraries are also available for LISP and Ada.

X runs under most UNIX systems, and MS-DOS, as well as several other operating systems. X display servers run on computers with either monochrome or color bitmap display hardware. The server distributes user input to client programs and accepts output requests from the various clients. The client programs can be located either on the same machine or elsewhere on the network.

For more information refer to Scheifler and Gettys' *X Window System, Second Edition*, O'Reilly & Associates, Inc.'s *X Window System Series*, Douglas Young's *The X Window System: Programming and Applications with Xt*, and SCO's *X Tool Kit* and *X Library* manuals.

xdata.h header file

The `<xdata.h>` header file specifies type definitions and routines used when changing the executable status of a data region.

See also "Header file" in this *Encyclopedia* and the manual pages for `execseg(S)` and `unexecseg(S)`.

XENIX cross development

The SCO UNIX Development System offers a number of tools and resources for developing XENIX applications under the UNIX operating system. Many of the routines and system calls available for the UNIX operating system are also available for XENIX. The `Routines(S)` manual page lists all of the routines and system calls in section (S) and contains a code indicating whether the routine works under the XENIX System V, the UNIX operating system, or both.

Compiling a cross developed application

After creating the XENIX application, compile it with the one or more of the following `cc(CP)` utility options:

Option	Description
-coff	COFF-format object
-xout	OMF-format object
-x3.2	SCO UNIX Release 3.2* compatible
-x2.3	XENIX System V compatible
-xenix	Pre-2.3 XENIX compatible (OMF objects only)
-compat	Ancient XENIX (circa XENIX 3.0) compatible (OMF objects only)
-M0	8086 processor
-M2	'286 processor
-M3	'386 processor
-Ms	'286 small model
-Mm	'286 medium model
-Ml	'286 large model
-Mh	'286 huge model
-Mc	'286 compact model

OMF and COFF differences

The main differences between the OMF (*Object Module Format*) (q.v.) and the COFF (*Common Object File Format*) (q.v.) are:

- OMF does not support shared libraries
- File locking (see description following list)
- MASM only produces OMF objects; **as**(CP) creates COFF objects

On releases before XENIX release 2.3, *file and record locking* (q.v.) was always mandatory for OMF. For release 2.3, use of the **cc -Fa** option for OMF files makes locks advisory, or the **-Fm** makes locks mandatory. When the **-x3.2** option is used, the **setgid** bit can cause an OMF file to have advisory locking.

For COFF files, locking is advisory unless a contention occurs with an OMF mandatory lock. Then the COFF lock becomes mandatory as well. For more information, refer to "Incompatibilities between COFF and OMF/x.out binaries" in the "COFF (Common Object File Format)" article in this *Encyclopedia*.

XENIX development utilities

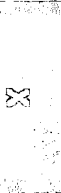
The section (XNX) describes the utilities for working with a compiled application. The utilities are:

Utility	Description
ar (XNX)	Maintains archives and libraries
hdr (XNX)	Display selected parts of an object file
ld (XNX)	Invokes the link editor
nm (XNX)	Displays a name list
prof (XNX)	Displays profile data generated by monitor (S)
ranlib (XNX)	Converts libraries to random libraries
size (XNX)	Displays the size of an object file
strip (XNX)	Removes symbols and relocation information

XENIX cross development

Note that in the remainder of this article, XENIX 2.3 refers to the version of XENIX prior to XENIX System V Version 2.3. References to XENIX System V in this chapter include all releases of System V. Consequently, references to XENIX 2.3 in this article refer to the pre-System V release numbering.

The remainder of this article chapter lists some of the differences among XENIX 2.3, XENIX 3.0, UNIX Version 7, UNIX System 3.0 and XENIX System V (all releases). The following sections are designed for users who wish to convert system calls in existing application programs for use on other systems.



Executable file format

XENIX 3.0, UNIX System 3.0, and some releases of XENIX System V execute only those programs with the *x.out* executable file format. The format is similar to the old *a.out* format, but contains additional information about the executable file such as:

- text and data relocation bases
- target machine identification
- word and byte ordering
- symbol table
- relocation table format
- the revision number of the kernel which is used during execution to control access to system functions

Release 2.3 of XENIX System V allows execution of COFF binaries as well as XENIX *x.out* executables. COFF binaries developed under this implementation of System V can execute under XENIX System V release 2.3 as well.

XENIX System V uses *little-endian* (low order word first in memory) word order for *longs*, whereas some XENIX 3.0 systems use *big-endian* (high order word first in memory) word order. XENIX System V checks the *x.out* header for information about the word order. XENIX System V maintains full XENIX 3.0 binary compatibility. XENIX System V executes XENIX 3.0 word-swapped (big-endian) executable files as well as XENIX 3.0 and XENIX System V (little-endian) executables.

Revised system calls

Some system calls in XENIX System V and UNIX System V revised and are different than the corresponding calls in previous systems. To provide compatibility for old programs, XENIX System V and UNIX System V maintain both the new and the old system calls and automatically check the revision information in the *x.out* header to determine which version of a system call should be made.

Version 7 additions

XENIX System V maintains several XENIX 3.0 and UNIX V7 features that were dropped from UNIX System 3.0. In particular, XENIX System V continues to support the `dup2(S)` and `ftime(S)` functions. The `ftime` function, used with the `ctime(S)` function, provides the default value for the timezone when the `TZ` environment variable has not been set. This means a binary configuration program can be used to change the default timezone. No source license is required.

Changes to the ioctl function

XENIX 3.0 and UNIX System 3.0 have full sets of XENIX 2.3-compatible `ioctl` calls. Furthermore, XENIX 3.0 and XENIX System V have resolved problems that previously hindered UNIX System 3.0 compatibility. For convenience, XENIX 2.3-compatible `ioctl` calls can be executed by a UNIX System 3.0 executable. The available XENIX 2.3 `ioctl` calls are:

<code>TIOCSETP</code>	<code>TIOCFLUSH</code>	<code>TIOCEXCL</code>
<code>TIOCGETP</code>	<code>TIOCSETD</code>	<code>TIOCHPCL</code>
<code>TIOCGETC</code>	<code>TIOCSETN</code>	<code>TIOCGETD</code>
<code>TIOCNXCL</code>	<code>TIOCSETC</code>	

Pathname resolution

If a null pathname is given, XENIX 2.3 interprets the name to be the current directory, but UNIX System 3.0 considers the name to be an error. XENIX 3.0 and XENIX System V use the version number in the `x.out` header to determine what action to take.

If the symbol `".."` is given as a pathname in a root directory that has been defined using the `chroot(S)` function, XENIX 2.3 moves to the next higher directory. XENIX 3.0 also lets the `".."` symbol move the current working directory to the next higher directory, but restricts its use to the super user. XENIX System V does not let the `".."` symbol move the current working directory to the next higher directory.

Using the mount and chown functions

XENIX 3.0 and UNIX System 3.0 restrict the use of the `mount(S)` system call to the super user. XENIX System V does not restrict the use of the `mount` system call. However, usually the `mount(C)` function is only executable by the super user. XENIX System V, XENIX 3.0, and UNIX System 3.0 let the owner of a file use the `chown(S)` function to change the file ownership.

Super block format

XENIX System V, UNIX System 3.0 and UNIX System V have different super block formats. XENIX System V and XENIX 3.0 use the System V format, with a different magic number for each revision. The XENIX System V and XENIX 3.0 super block each has an additional field at the end that can be used to distinguish between XENIX 2.3, 3.0 and System V super blocks. XENIX System V and XENIX 3.0 check this magic number at boot time and during a mount. If a XENIX 2.3 super block is read, XENIX 3.0 converts it to the new format internally. Similarly, if a XENIX 2.3 super block is written, XENIX 3.0 converts it back to the old format. This permits XENIX 2.3 kernels to be run on file systems also usable by UNIX System 3.0.

However, XENIX System V is word-swapped relative to XENIX 3.0. Even though the super block formats are the same, the order of bytes in long words is different. XENIX System V's **mount(C)** and **fsck(C)** commands cannot be used to mount or check XENIX 3.0 filesystems.

Separate version libraries

XENIX System V supports the construction of XENIX 3.0 executable files. This system maintains both the new and the old versions of system calls in separate libraries.

XENIX operating system

The following table summarizes the XENIX release dates.

XENIX 286 and XENIX 386 release dates

Release	Version	XENIX 286	XENIX 386
2.1	3	Winter 1986	—
2.2	0	Summer 1986	—
2.2	1	Summer 1987	Summer 1987
2.2	2	—	Winter 1988
2.2	3	Summer 1988	Spring 1988
2.3	1	—	Fall 1988
2.3	2	Summer 1989	Winter 1989

The UNIX System V/386 Development System supports cross-development for the XENIX operating system.

X/Open portability guide (XPG)

The *X/Open Portability Guide* defines a Common Applications Environment that is built on the interfaces from the POSIX standard and discusses other aspects required of a comprehensive applications interface.

X/Open is not a standards organization, but rather a consortium of information system suppliers who have agreed to support the standard as it evolves, to promote greater source code portability for applications code.

See the “Bibliography” in this *Encyclopedia* for a list of all the volumes in this set, and the manual pages in section (S) for XPG compliance details. Users who want to write code that conforms with XPG should use the **-xpg3** option to the **cc(CP)** command.

x.out.h header file

The `<sys/x.out.h>` header file specifies routines and files to be included that describe the executable file created automatically on XENIX operating systems for *XENIX cross development* (q.v.). The *x.out* format is often referred to as the *OMF (Object Module Format)* (q.v.).

See also "Header file" in this *Encyclopedia*.

XTI (X/Open Transport Interface)

XTI is X/Open's extension of *TLI* (q.v.), AT&T's Transport Layer Interface (TLI). It provides a protocol-independent, transport-layer interface for networking applications. XTI is specified in the *X/Open Portability Guide 3 Volume 7*. It consists of a series of C language functions that are based on the functions in TLI, which in turn is based on the transport service definition for the *OSI (Open Systems Interconnection)* (q.v.) model.

XTI provides access to the transport mechanism whereby data moves from one computer to another. Applications can link endpoints synchronously via a direct connection or in a connectionless mode, as discussed in the "TLI" article. Each data transfer makes use of two transport endpoints, one on the source computer and one on the destination computer. These endpoints are identified by a file descriptor on each computer, and may support only one transport connection at a time.

XTI provides these basic enhancements to TLI:

- Modifications to the interface to handle previously-existing performance and protocol problems.
- The removal of dependencies on specific UNIX operating system versions and specific transport providers, such as ISO. In particular, this release of the XTI library provides support for the TCP and UDP transport providers.
- An extended set of function error types that further simplify network and application troubleshooting.

Implementation definition

The XTI library supplied with this release of the Development System implements all XTI functions specified in the *X/Open Portability Guide 3 Volume 7* with the following implementation details:

- The optional `t_optmngmt` function is not supported.
- If the application communicates over TCP/IP and passes options with `struct tcp_options` (defined in the `<xti.h>` header file), the XTI library makes a formal check on the options, but does not pass them to the TCP/IP provided.
- XTI returns a `tcp_options` structure with every field set to `T_UNUSED` for `t_connect(S)`, `t_listen(S)`, or `t_rcvconnect(S)`.

See the Release Notes for a list of deviations that apply when the transport provider uses the `STREAMS` interface.

Converting from TLI to XTI

The TLI and XTI libraries contain the same routines, and most applications are coded the same whether they are compiled with TLI or XTI. An application that uses the TLI functions can be converted to use the XTI functions as follows:

- Use the `<xti.h>` header file rather than `<tiuser.h>`.
- Make small modifications to the code as needed, because XTI specifies some items that TLI does not and makes enhancements to functions that affect their functionality.
- Compile the code with the `-lxti` or `-lxti_s` option rather than the `-lnsl` or `-lnsl_s` option.

TLI does not specify as many implementation details as XTI does; many of the issues that would affect a port of code from one TLI implementation to another also impact the port from TLI to XTI. The major issues here are device path names, network address formats, and protocol.

Device path names

The `t_open(S)` routine establishes a transport endpoint to begin access to a network service. One of the arguments to `t_open` is the *path* of the network device that is the transport provider. For example, UNIX System V/386 uses `/dev/inet/tcp` as the path for the TCP/IP implementation. Neither TLI nor XTI specify what this *path* argument is. Most applications that run under the TLI implementation running on SCO UNIX System V/386 will use the same path for XTI, but applications being ported from other TLI implementations may require a different *path* argument to `t_open`.

Network address formats

The `tbind(S)` routine is called after `t_open` to associate a Transport Connection End Point with a Protocol Address. This “local transport address” serves as the “transport service access point” (TSAP) that other programs use to access this program much as your phone number is used by people who want to access you at your phone. TLI does not define the address syntax, so each network implementation can define its own address syntax. XTI explicitly defines this address format, with one format used for TCP/IP protocols such as TCP and UDP and another format used for ISO protocols.

The second and third arguments to `t_bind` are `struct t_bind` structures that specify the protocol address of the caller and the protocol address that is actually returned by `t_bind`. The `t_bind` structure and the `netbuf` structure it refers to are defined in the `<xti.h>` header file as follows:

```
struct t_bind {
    struct netbuf  addr;
    unsigned      qlen;
};

struct netbuf {
    unsigned int   maxlen;
    unsigned int   len;
    char          *buf;
};
```

The `buf` element of the `netbuf` structure within the `t_bind` structure is a pointer to a buffer which contains the address. The format of this buffer and its exact contents are not specified in TLI. When using TLI with SCO TCP/IP, this buffer should look exactly like the `sockaddr_in` structure that is defined in our TCP/IP documentation and header files. Other TCP/IP implementations may use other structure definitions. XTI explicitly defines what this buffer looks like.

Other TLI and XTI functions that use network addresses (such as `t_accept(S)`, `t_call(S)`, `t_listen(S)`, `t_rcvconnect(S)`, `t_rcvudata(S)`, `t_rcvuderr(S)`, `t_snddis(S)`, and `t_sndudata(S)`) also use network addresses in some way and need to know this address structure just as `t_bind(S)` does.

Protocol options

A network protocol implementation allows the user of the service to affect the operation of the network connection in certain specified ways, usually code with the `t_optmgmt(S)` call. TLI does not specify the format and meaning of protocol options. As with network addresses, the arguments to `t_optmgmt(S)`, `t_accept(S)`, `t_connect(S)`, `t_listen(S)`, `t_rcvconnect(S)`, `t_rcvudata(S)`, `t_rcvuderr(S)`, `t_snddis(S)`, and `t_sndudata(S)` contain a `struct netbuf` that refers to some buffer that is supposed to contain option management data of some sort. TLI leaves this buffer's definition undefined, while XTI explicitly defines which options are supposed to be supported with TCP/IP and ISO protocol services, and how they are specified.

Error return codes

XTI specifies different return error codes than TLI does for some functions. Most error returns for most functions are the same, but most functions also have at least one error return that is different in XTI and TLI. In general, the XTI error codes better identify the causes of errors than does TLI.

Programs that do not explicitly check for particular error conditions but instead use `t_error(S)` or a similar convention to print out a message are unaffected. If your program checks for specific error conditions, refer to the XTI documentation carefully to ensure that the error conditions for which the code checks are supported.

State differences

Both the TLI and XTI interfaces specify a series of defined states through which the application program and the protocol stack proceed. The exact progression varies depending on what the application program, the network, and the application program on the remote end of the connection and the order in which they do it, but the order is likely to be different under XTI than it was under TLI.

Most application programs are unaffected by this, but programs that use the asynchronous mode of operation for TLI or XTI must explicitly deal with correct state changes. Compare the states assumed in your program with those defined for XTI to ensure that they match.

XTI restrictions

Be aware of the following when developing applications that use XTI:

- Applications developed in strict conformance to XTI may not immediately function over a transport provider written in conformance to TLI. This is because XTI uses only a subset of TLI functions and facilities, and because several TLI functions were modified when incorporated into XTI.
- XTI does not permit an application to wait for incoming connections without specifying the remote protocol address. This functionality is the same under TLI.
- Neither XTI nor TLI handle management aspects of the interface, such as portable addressing or protocol characteristic selection. If you want this functionality, you must obtain it from a higher-level management function.

XTI state guidelines

The following list contains general guidelines for maintaining the correct state of the XTI interface. If your application forces XTI into an incorrect state, XTI aborts.

- The transport provider, such as TCP or UDP, must at all times keep track of the state of the user interface. For example, the provider must know if data is currently being transferred, or if an endpoint is idle.
- The transport provider should never process a function that places the interface out of state.
- If a function is processed out of sequence, the function returns an error and the state remains unchanged. For example, if an attempt is made to transfer data while the interface is not in the T_DATAXFER state, that data is not accepted or forwarded by the transport provider, and the state of XTI remains as it was before the data transfer was attempted.
- Each transport endpoint begins in the uninitialized state, T_UNINIT. It must be initialized and bound before the transport provider can view it as active.
- The uninitialized state is also the final state for each endpoint. The transport provider may not re-use a transport endpoint until it is closed by the t_close(S) function.
- If you are using connection-oriented mode, you can initiate the transport connection release phase at any time during the connection establishment phase or data transfer phase.

xti.h header file

The `<xti.h>` header file defines the C structures and constants for the XTI (q.v.) library. Use this header file rather than the `<tiuser.h>` header file to convert an application that uses the TLI (*Transport Layer Interface*) (q.v.) functions to use the XTI functions. This file is AT&T SVID Issue 2 and X/Open XPG3 compliant.

Y

yacc

The yacc (yet another compiler compiler) program is a parser-generator. The yacc user prepares a specification that includes a set of rules to describe the syntax of the input, code to be invoked when a rule is recognized, and a *lexical analysis* (q.v.) routine responsible for mapping input characters to tokens. The input to yacc is similar to the Backus-Naur Form (BNF) notation.

yacc reads a specification file and turns it into a C language function that implements a parser. When executed, this routine invokes the lexical analyzer to pick up items from the input stream. The parser reads tokens and compares them to the input construct rules, called grammar rules. When one of the rules is recognized, the user code supplied for this rule (an action) is invoked. Actions are fragments of C language code; they can return values and make use of values returned by other actions.

The **specification** file describes a context-free grammar. A context-free grammar indicates the hierarchical relationships of language constructs; it can be divided into four basic parts: terminal symbols (tokens), non-terminal symbols (syntactical elements), definitions of nonterminal symbols (production rules), and a start rule which represents the highest-level element of the language to be parsed. yacc converts a context-free grammar into a set of tables for a simple automaton that implements a parser for the grammar.

For more information about yacc, see yacc(CP) and O'Reilly & Associates' *lex* & yacc. See also "Lexical analysis" and "Parsing" in this *Encyclopedia*.

Z

Zombie state

A zombie process (also called a defunct process) is a *process* (q.v.) that has terminated its execution, but whose parent has not yet executed the **wait** system call to collect its completion status. All the memory held by the process is discarded, but the zombie process still occupies a slot in the *kernel's* (q.v.) process table until the parent process calls **wait**. If the parent terminates before calling **wait** the child process is inherited by process 1, known as **init**.

The presence of zombie processes can indicate a problem with the parent process. If a problem with the parent process prevents it from acknowledging the termination of its children, zombie processes may eventually fill up all the slots in the process table, so that new processes cannot be created. Killing the parent process usually restores the system to sanity.

0-9

80x86-family processors

The Intel Corporation manufactures a group of processors referred to as the 80x86-family of processors. The 80x86-family of processors includes: the 8088 and 8086, the 80186 and the 80188, the 80286, the 80386, and the 80486. All of the processors in the family have several features in common: memory is organized using a segmented architecture; the instruction set is upwardly compatible — all features available in the early versions of the processor are also available in the newer versions, but the new versions contain additional features not supported in the old versions; the register set is also upwardly compatible.

An important distinction between members of the 80x86-family is mode of operation; members can operate in *real mode* or *protected mode*. On a processor running in real mode, a process can access all parts of physical memory. On a processor running in protected mode, a process can access only those parts of physical memory which have been allocated to it.

Associated with some members of the 80x86-family of processors is a math coprocessor. These include: the 8087, the 80287, and the 80387. The math coprocessors work in conjunction with the 80x86 family of processors to perform calculations faster than can be done with the 80x86 processors alone. If these processors are not present, the operating system or application must emulate floating point operations in software.

A summary of the characteristics of the different processors follows:

8088 and 8086

These processors work in real mode. They are designed to run a single process. The processor can address up to one megabyte of memory. Addresses specified in *assembly language* (q.v.) correspond to physical memory addresses.

The 8088 uses an 8-bit data bus, and the 8086 uses a 16-bit data bus. This makes the 8086 faster. However, from the programming standpoint, the two processors are identical except that the 8086 handles certain data more efficiently if it is word-aligned. In an assembly language program, data must be explicitly word-aligned with the *EVEN* or *ALIGN* directives. (See the *Macro Assembler Writer's Guide* for more information on these directives.) In a C language program, the compiler automatically word-aligns data as appropriate.

80186 and 80188

These processors are identical to the 8086/8088 except that new instructions have been added and some old instructions have been optimized. They run significantly faster than their predecessors.

80286

This processor has the added instructions and speed of the 80186. It can run in the real mode of the 8088 and 8086, but it also has an optional protected mode in which multiple processes can be run concurrently.

In protected mode, the processor can address up to 16 megabytes of memory. However, when memory is accessed in protected mode, the addresses may not correspond to physical memory. In protected-mode, the operating system manages memory dynamically. Additional privileged instructions for initializing protected mode and controlling multiple processes are available.

80386

This processor is fully compatible with the 80286 but, at the system level, it implements many new features including: virtual memory, multiple 8086 processes, and addressing for up to four gigabytes of memory.

For the applications programmer, the 80386 supports all the instructions of the 80286 and some additional instructions. It also allows use of 32-bit registers and addressing modes. The 80386 also operates significantly faster than the 80286.

80486

This processor is basically a faster 80386 processor along with a built-in 80387 math coprocessor. The 80486 includes additional functionality as well.

NOTE Processors in the 80x86 family of processors are often referred to using only the last three digits of the processor name. For example, the Intel 80386 is often referred to as simply the 386.

For more information on each of these processors, refer to the appropriate chip's programmer's reference manual published by Intel. Refer to the cc(CP) manual page for more information about memory models on processors.

Bibliography

The following commercial books and articles provide details about topics overviewed in this encyclopedia. This list is provided for reference only; it is not comprehensive and The Santa Cruz Operation does not guarantee the accuracy of these publications.

Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Reading, MA: Prentice-Hall, 1988. Written by the developers of the `awk(C)` language, this book describes `awk`'s functionality and gives guidelines for using this versatile tool.

Arthur, Jay. "Programmer Productivity and the Unix [sic] System." *UNIX/World* (July 1986): 26-33. Describes how to increase programmer productivity using the UNIX tools.

AT&T Bell Laboratories. *UNIX System Readings and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987. Reprints of the July-August 1978 and October 1984 issues of the *AT&T Bell Laboratories Technical Journal* containing articles about the UNIX operating system by early developers. These articles help the user to understand the historical development of the UNIX Operating System.

AT&T. *STREAMS Primer* and *STREAMS Programmer's Guide*. Englewood Cliffs, NJ: Prentice-Hall, 1987. AT&T's original documents which explain the STREAMS interface. The UNIX System V/386 Release 3.2 implementation is identical to the AT&T implementation, although AT&T supports some non-STREAM kernel functions that are not supported on UNIX System V/386 Release 3.2.

Bach, Maurice J. *The Design of the UNIX Operating System*. Englewood Cliffs, NJ: Prentice-Hall, 1986. A technical discussion of the internals of the UNIX System V Operating System, written shortly before System V, Release 3. Somewhat outdated, but still very useful.

Banahan, Mike and Andy Rutter. *The UNIX Book*. New York, NY: John Wiley and Sons, 1983. A general guide to the features and capabilities of the UNIX Operating System.

Bent, Wilson H. Jr. "Compiling Stubborn Programs." *UNIX/World* (November 1987): 113-122. Describes strategies for getting difficult software to compile.

- Bentley, Jon Louis. *Programming Pearls*. Reading, MA: Addison-Wesley, 1986.
- Bentley, Jon Louis. *More Programming Pearls: Confessions of a Coder*. Reading, MA: Addison-Wesley, 1988.
- Bentley, Jon Louis. *Writing Efficient Programs*. Englewood Cliffs, NJ: Prentice-Hall, 1982. Discusses numerous programming practices that improve the efficiency of programs. While based on PASCAL, the concepts are applicable to all high-level programming languages.
- Bolsky, Morris I. and David G. Korn. *The Korn Shell Command and Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1989. This book is the specification of the Korn shell language and a reference handbook for `ksh`, the program that implements the Korn shell language. Contains a tutorial for using `ksh` as an interactive command language and as a programming language. Includes examples illustrating the features of `ksh`, and many chapters have exercises. Also contains a quick reference summary of the Korn shell language.
- Bourne, S. R. *The UNIX System*. Reading, MA: Addison-Wesley, 1982.
- Christian, Kaare. *The UNIX Operating System*. New York, NY: Wiley-Interscience, 1983.
- Date, C.J. *An Introduction to Database Systems*, Vol. 1, 5th printing. Reading, MA: Addison-Wesley Pub. Co., 1990. Provides an introduction to database theory, design, and implementation.
- Dougherty, Dale, "AWK: A Not So Awkward Tool." *UNIX/World* (May 1987): 83-89; (June 1987): 81-89; (July 1987): 83-88. A three-part series that describes the `awk(C)` command for processing lists and tables, developing applications, and acronym list manipulation.
- Ellis, M. and Stroustrup. *The Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley, 1990.
- Gauthier, R. *Using the UNIX System*. Reston, VA: Reston Publishing Company, 1981.
- Gircys, Gintaras R. *Understanding and Using COFF*. (Nutshell Series). Sebastopol, CA: O'Reilly & Associates, 1988.
- Gofton, Peter W. *Mastering UNIX Serial Communications*. San Francisco, CA: Sybex, 1990. An introduction to serial communication facilities on SCO XENIX and UNIX systems. Includes a section for programmers with chapters entitled "System Calls to Access Serial Devices" and "Programming Tips and Techniques."

Goodheart, Berny. *UNIX Curses Explained*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

Harbison, Samuel P. and Guy L. Steele. *C: A Reference Manual*, second edition. Englewood Cliffs, NJ: Prentice-Hall, 1987. An explanation of the C language. Includes an index and cross-references within the text.

Kernighan, Brian W. and P.J. Plauger. *Software Tools*. Reading, MA: Addison-Wesley, 1976.

Kernighan, Brian W. and P.J. Plauger. *The Elements of Programming Style*. New York, NY: McGraw Hill, 1978.

Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*, second edition. Englewood Cliffs, NJ: Prentice-Hall, 1988. The first edition of this book is the definitive definition of the C programming language. The second edition describes ANSI version of the C programming language, which is provided with Microsoft's 6.0 C compiler that is called by the cc(CP) command in the 3.2.3 Development System software (see Tondo for a related book).

Kernighan, Brian W. and Rob Pike. *The UNIX Programming Environment*. Englewood Cliffs, NJ: Prentice-Hall, 1984. Introduces the fundamental elements of the traditional UNIX programming environment, including the filesystem, the shell and shell programming, filters, using standard I/O, and system calls. It also includes chapters on program development and document preparation.

Koenig, Andrew. *C Traps and Pitfalls*. Reading, MA: Addison-Wesley, 1988.

Lapin, J.E. *Portable C and UNIX Programming*. UNIX Prentice-Hall, 1987. Gives a history of UNIX operating systems and summarizes the major differences between the different versions. Also discusses the general issues involved in portability and gives specific suggestions for writing portable C code. Written before System V, Release 4, and SUNOS were available, the general information covered is useful.

Lewine, Donald. *POSIX Programming Manual*. Sebastopol, CA: O'Reilly and Associates, 1991. This book contains two separate parts: a programming guide and a reference guide that present information about the POSIX standard for application programmers. The programming guide section presents an overview of developing POSIX applications and is also appropriate for programmers who are new to the UNIX Operating System and need to learn the application interface. The reference guide lists all POSIX library functions in alphabetical order, all data structures and their members, all error codes, and standard header files and the information they define.

Digitized by Google

- Lomuto, Ann Nicols and Nico Lomuto. *A UNIX Primer*. Englewood Cliffs, NJ: Prentice-Hall, 1983. Beginner's guide to the UNIX Operating System.
- Mason, Tony and Doug Brown. *lex & yacc* (Nutshell Series). Sebastopol, CA: O'Reilly & Associates, 1990.
- McGilton and Morgan. *Introducing the UNIX System*. New York, NY: McGraw-Hill, 1983.
- Miller, Chris D. F. and R. D. Boyle. *UNIX for Users*. Rockville, MD: Blackwell Scientific Publications, 1984.
- O'Reilly, Tim. *X Toolkit Intrinsics Reference Manual, Volume 5*. (X Window Systems). Sebastopol, CA: O'Reilly & Associates, 1990.
- Plum, Thomas, and Jim Brodie. *Efficient C*. Cardiff, NJ: Prentice-Hall, 1985.
- Plum, Thomas. *Learning to Program in C*. Cardiff, NJ: Prentice-Hall, 1983. Provides an introduction to (or refresher on) the C programming language.
- Plum, Thomas. *C Programming Guidelines*. Englewood Cliffs, NJ: Prentice-Hall, 1984. Organized in reference manual fashion, this book suggests programming practices that lead to C code that is reasonably portable and maintainable.
- Rochkind, Marc. *Advanced UNIX Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1985. Provides information and examples for using the classic UNIX system calls. Many system calls have been added since this book was published, but this introduces the basics.
- Schreiner, Axel T. and H. George Friedman, Jr. *Introduction to Compiler Construction with UNIX*. Englewood Cliffs, NJ: Prentice-Hall, 1985. Describes *lex*(CP) and *yacc*(S).
- Sobell, Mark G., *A Practical Guide to UNIX System V*. The Benjamin/Cummings Publishing Company, Inc 1985. Describes the UNIX operating system and Bourne shell programming.
- Stevens, W. Richard. *UNIX Network Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1990. Describes how to write applications that run over the various network protocols supported on UNIX operating systems. In addition to basic networking concepts, contains many code examples illustrating how to write applications that use STREAMS, sockets, Interprocess Communications facilities, Transport Level Interface, Internet, and File Transfer Protocol.

Stroustrup, Bjarne. *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1987.

Tondo, Clovis L. and Scott E. Gimpel. *The C Answer Book*, second edition. Englewood Cliffs, NJ: Prentice-Hall, 1985. Gives solutions to the exercises in the Kernighan/Ritchie *C Programming Language* book. The 1985 edition accompanies the 1978 Kernighan/Ritchie.

X/Open Company Limited, *X/Open Portability Guide*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

Volume 1: XSI Commands and Utilities

Volume 2: XSI System Interface and Headers

Volume 3: XSI Supplementary Definitions

In addition to the supplementary definitions (a basic glossary of terms used throughout the XPG), defines methods for implementing internationalization, the curses(S) interface, and source code transfer methods.

Volume 4: Programming Languages

Volume 5: Data Management

Volume 6: Window Management

Volume 7: Networking Services

Young, Douglas A. *The X Window System®: Programming and Applications with Xt, OSF/Motif® Edition*. Englewood Cliffs, NJ: Prentice-Hall, 1990. Describes the concepts of the X Window System and OSF/Motif, and describes how to write application programs using these libraries. Includes numerous code examples to illustrate the principles discussed.

Weiner, Richard S. and Lewis Pinson, *Introduction to Object Oriented Programming and C++*. Reading, MA: Addison-Wesley, 1988.

ENCLOSURE

Index

Special characters

/ in pathname, 201

A

Absolute file, 189
 Absolute path, 201
 Access permissions, 236
 acct.h header file, 7
 adb, using in debugging, 55
 Address mapping, virtual-to-physical, 296
 Address Resolution Protocol, 273
 Address space, user, 290
 agent.h header file, 7
 Aho, Alfred V., 317
 ANSI (American National Standards Institute)
 C language standard, 8
 C standard, 27
 defined, 7
 ANSIC
 standard, 257
 stringizing, 262
 a.out file, 8, 149, 160
 a.out.h header file, 8, 39
 aouthdr.h header file, 8
 Application
 error handling, 85
 program, 9
 Archive library
 defined, 9
 shared, 243
 with link editor, 160
 Archive library, 13
 argc
 defined, 11
 usage example, 84
 Argument
 check with function prototype, 107
 command line, 35
 defined, 10
 function variable number, 107
 in data segment, 215
 shell script, 11
 white space delimiter, 297

argv
 defined, 11
 usage example, 84
 ar.h header file, 13
 ARP, 273
 ARPANET, 47
 Arthur, Jay, 317
 asctime(S)
 date format, 49
 discussion, 276
 Assembler, defined, 15
 Assembly language
 defined, 15
 history, 116
 assert.h header file, 15
 AT&T
 Portable C Compiler, 27
 Portable C compiler, 40
 System V Interface Definition, 269
 audit database, 280
 Audit routines, 281
 audit.h header file, 15
 Auditing system access, 237
 audit(S), 282
 auth directory in /etc, 280
 auth hierarchy in /tc/ files, 280
 authaudit(S), 281
 authcap(S), 282
 awk
 bibliographic entry, 317-318

B

Bach, Maurice J., 317
 Bad track scans, 235
 Banahan, Mike, 317
 BASIC, defined, 17
 BCD (Binary Coded Decimal), 18
 bcopy, 17
 Bent, Wilson H. Jr., 317
 Bentley, Jon Louis, 318
 Binary
 code, 254
 compatibility, 17
 defined, 17
 format, 189

- Block device driver, 19
- Block mode display processing, 273-274
- Block size, filesystem, 265
- Blocked process, 214
- Blocked signal, 250
- Bolsky, Morris I., 318
- bool.h header file, 19
- Bourne, S. R., 318
- Bourne shell, 20
- Boyle, R.D., 320
- Brodie, Jim, 320
- Brown, Doug, 320
- Browser, cscope, 43
- BSIZE manifest constant, 20
- buf structure, 21
- Buffer
 - defined, 21
 - manipulation routines, 22
- buf.h header file, 21
- Bugs, detecting with lint, 162
- Byte
 - defined, 22
 - order, 23
- bzero, 17

C

- C++
 - bibliographic entry, 318, 321
- C compiler
 - preprocessor, 27
 - type size, 285
- C manifest constants, 26
- C preprocessor, 27, 53
- C programming language
 - ANSI specification, 8
 - defined, 27
- C shell, 27
- C1 and C2 security levels, 236
- Calculating large numbers, 18
- Calendar
 - Gregorian, 50
 - Julian, 50
 - date, 49
 - leap year, 50
- call_list and callsave structure, 162
- Canonical
 - processing, 29
 - queue, 30
- Caught signal, 250

- cc options for XENIX cross development, 300
- cc(CP)
 - M option, 15
 - Zp option, 263
 - ansi option, 8
 - d option, 40
 - ldbm option, 48
 - libraries used, 157
 - o option, 8
 - posix option, 213
 - svid option, 270
 - xenix option, 8, 48
 - xpg option, 304
- CGA (Color Graphics Adapter), 111
- Channels, DMA, 67
- Character classification and conversion, 31
- Character device driver
 - defined, 32
 - interface, 226
- Characters, special, 227
- Child process
 - debugging, 54
 - defined, 32
 - file descriptors, 95
 - file pointers, 100
 - inherited attributes, 215
 - memory structures, 178
- chmod(C), 202
- chmod(S), 202
- Christian, Kaare, 318
- Circumflex, regular expression, 227
- Clients, X Windows, 299
- Clock
 - system, 34
 - time, 276
- COBOL, defined, 34
- Code
 - binary, 254
 - object, 189
 - source, 254
- CodeView, using in debugging, 54
- COFF, understanding and using, 318
- COFF (Common Object File Format)
 - debugging, 54
 - filehdr.h header file, 97
 - ldfcn.h header file, 155
 - section, 236
 - section header, 233, 268
 - symbol table, 258
- COFF (Common Object File Format), 35

- coffconv, 37
- Command
 - argument, 10
 - defined, 35
 - syntax, 269
- Command line
 - argument, shell script, 11
 - argument storage order, 11
 - debugging with sdb, 55
 - defined, 10
 - delimiter, 55
 - white space delimiter, 297
- Common Applications Environment
 - XPG/3
- Compact memory model, 179
- Compiler
 - BASIC, 17
 - C language, 27
 - COBOL, 34
 - FORTRAN, 105
 - Portable C compiler, 40
 - optimization, 190
- Compiling a XENIX application, 300
- Connectionless-mode, defined, 278
- Connection-mode, defined, 277
- Context switch, 40
- Context-free grammar, 311
- Conversion
 - character, 31
 - data, 47
- Cooked data, 30
- Copy-on-write scheme, 33
- Core file, 42, 53
- core.h header file, 39, 43
- cpp, preprocessor, 213
- CPU
 - allocation, 233
 - context switch, 40
- crash(ADM), inode command, 122
- creat(S), internal implementation, 99
- Cross compilers, 271
- Crypt libraries, 43
- cscope(CP), 43
- csh(C), set TZ variable, 287
- csh(C), 27
- ctime(S)
 - date format, 49
 - discussion, 276
- ctrace, using in debugging, 55
- CTS flow control, 102
- ctype.h header file, 43

- curses
 - explained, 319
 - window, 298
 - with termcap, 272
- curses, 44
- curses.h header file, 39, 46
- Cursor optimization
 - ETI, 88
- cvtomf, 37

D

- DARPA, 47, 272
- .data
 - reloc.h header file, 230
 - section, 214, 236
- Data pointers with memory models, 180
- Data segment, 215
- Data storage blocks, 100
- Data structure
 - queue_t, 223
 - region, 227
- Data symbol, 48
- Database management system
 - ISAM, 143
 - XENIX, 48
 - defined, 48
- Database manager, XENIX, 51
- Date
 - epoch, 49
 - international representations, 152
 - operating system, 49
- Date, C.J., 318
- daylight variable, \$TZ Environment Variable
- dbd (disk block descriptor), 50
- dbm.h header file, 51
- dbxtra, using in debugging, 54
- DC1 and DC3 flow controls, 102
- Debugging
 - defined, 51
 - using CodeView, 54
 - using adb, 55
 - using ctrace, 55
 - using dbxtra, 54
 - using printf(S), 52
 - using sdb, 55
- Defunct process, 313
- Delimiter, defined, 55
- Device, special files, 63
- Device driver, block, 19

Device drivers, defined, 56
 Diagrams, typesetting, 209
 dial.h header file, 64
 difftime(S), discussion, 276
 Directory
 access permissions, 203
 defined, 65
 dir.h header file, 67
 routines to control, 66
 dirent.h header file, 67
 dir.h header file, 67
 Display processing, block mode, 273-274
 divvy(ADM), 266
 DMA (Direct Memory Access), 67
 DMAEXCL tunable, 67
 dma.h header file, 68
 dmareq structure, 68
 Documentation
 commands, 35
 formatting, 279
 using manual pages, 171
 using permuted index, 173
 Domain, 68
 DOS cross-development, 179
 Dougherty, Dale, 318
 DSSSL (Document Style Semantics and Specification Language), 70
 DTD (Document Type Definition), 242
 Dynamic analysis, in profilers, 220
 Dynamic library, 71

E

Editor, link, 160
 Effective group ID, 202, 215
 Effective user and group ID's, 87
 Effective user ID, 202, 215
 EGA (Enhanced Graphics Adapter), 111
 EGID, 87
 Ellis, M., 318
 EMPTY macro, 162
 Encryption, defined, 73
 Endian order, 75
 environ variable, 81
 Environment, 81, 95
 Environment variable, 81, 293
 Epoch time, 49
 eqn
 preprocessor, 82, 213
 errno, returns from system calls, 269

errno.h header file, 83
 Error handling
 defined, 83
 errno.h header file, 83
 errorhandling, TLI and XTI, 275
 Error return codes
 XTI, 308
 /etc/auth directory, 280
 /etc/group file, 114
 /etc/magic file, 169
 eti.h header file, 87
 EUID, 87
 exec(S), 178
 Executable file, OMF, 189
 Executable file, 87, 189, 220
 Executable file format
 COFF, 35
 a.out file, 8
 exit, C function, 88
 exit(S), 33
 Exponential functions, 175
 Expression, regular, 227
 Extended Terminal Interface (ETI), 87-88, 105

F

fatal.h header file, 89
 fatal(S), 89
 fcntl.h header file, 39
 fcntl.s header file, 89
 Field
 defined, 89
 delimiter, 55
 File
 ISAM, 143
 Names, 97
 absolute, 189
 access permissions, 202
 access routines, 92
 access system calls, 92
 core, 53
 defined, 90
 device special, 63
 executable, 87, 189, 220
 inode, 120
 locking, 38
 object, 87, 160
 relocatable object, 189
 File access, buffered, 260

File and record locking
 UNIX system, 89
 defined, 93
 File descriptor, 94, 215
 File descriptor table, user area, 98
 File structure, 95
 File table, system, 96, 98
 filehdr.h header file, 97
 Filesystem access, inode table, 121
 Filesystem structure
 access tables, 98
 defined, 100
 ilist, 119
 inumber, 126
 super block, 264
 Filter, 101
 float.h header file, 102
 Floating point
 IEEE standard, 119, 176
 Flow control, 102
 Flush, 103
 flushq, example, 104
 fork(S), 33, 178, 219
 form.h header file, 105
 FORTRAN, defined, 105
 fprintf(S), example of, 84
 Friedman, H. George Jr., 320
 ftok(S), 139
 ftw.h header file, 105
 Function
 argument, 10
 defined, 106
 pointers with memory models, 180
 prototype, 107
 variable number of arguments, 107

G

Gauthier, R., 318
 getenv(S), 81
 Gircys, Gintaras R., 318
 gmtime(S), discussion, 276
 Gofton, Peter W., 318
 Goodheart, Berny, 319
 Grammar, context-free, 311
 grap, preprocessor, 213
 Graphics adapters, 111
 Gregorian calendar, 50
 group file in /etc, 114

Group ID
 effective, 87, 202
 real, 226
 Group identification, 215
 Group structure, 114
 Group user ID, 215
 grp.h header file, 114

H

Harbison, Samuel P., 319
 Hashing, 115
 Header file
 binary incompatibilities, 40
 defined, 115
 High-level language, 116
 Host adapter, 117
 Huge memory model, 179
 Hypermedia/Time-Based Structuring Language, 117
 HyTime, 117
 I18N, 119
 iBCS, 122
 ICANON, 30
 ICMP, 273
 IEC (International Electrotechnical Commission), 145
 IEEE floating-point standard, 176
 ieee.h header file, 119
 i_ermo.h header file, 119
 Ignored signal, 250
 ilist, 100, 119
 immu.h header file, 296
 Impure memory model, 179
 include directory in /usr, 115
 Include file
 Header file
 #include statement, 116
 Inode
 defined, 120
 super block, 265
 inode, pipes, 210
 Inode table, system, 98, 121
 Instruction segment, 215
 Intel binary compatibility specification. *See* iBCS
 Internationalization
 LANG environment variable, 151
 defined, 123
 langinfo.h header file, 152

Internationalization (*continued*)

- locale, 162
- locale.h header file, 165
- localization, 165
- message catalog, 176, 181
- internationalization, date and time, 49
- Internet addresses, 124
- Internet Control Message Protocol, 273
- Internet Protocol, 273
- Internet protocol, 47
- Interpreted language, 125
- Interpreter, BASIC, 17
- Interrupt
 - SCSI, 234
 - clock, 34
 - context switch, 40
 - defined, 126
- Intrinsics, 126
- inumber, 121, 126
- I/O subsystem, 127
- ioctl, termio structure access example, 30
- ioctl routines, 127
- IPC facilities
 - defined, 138
 - messages, 182
 - shared memory, 39, 246
- ipcs(ADM), 139
- ISAM files
 - database management, 48
 - defined, 143
 - file and record locking, 94
 - record handling, 226
- isam.h header file, 145
- ISO 8072, 195
- ISO (International Organization for Standardization), 145

J

- Julian calendar, 50

K

- Kernel
 - I/O subsystem, 127
 - defined, 149
 - documentation of functions, 172
- Kernighan, Brian W., 317, 319
- kill(S), 219
- Koenig, Andrew, 319

- Korn, David G., 318
- Korn shell, command and programming language, 318
- Korn Shell, 149
- Ksh(C), defined, 149
- ksh(C), set TZ variable, 287

L

- LANG environment variable, 151, 287
- langinfo.h header file, 152
- Languages, non-English, message catalog, 181
- Lapin, J.E., 319
- Large memory model, 179
- ldfcn.h header file, 155
- Leap year, calculating, 50
- Lewine, Donald, 319
- lex, bibliographic entry, 320
- lex, 156
- Lexical analysis, 156, 311
- .lib section, 243
- Library
 - archive, 9, 13, 160
 - common C, 158
 - dynamic, 71
- Library routines
 - defined, 157
 - documentation, 172
- limits.h header file, 159
- Line discipline, 30, 283
- linenum.h header file, 160
- Link editor, 160
- Link kit, 161
- Linking
 - memory models, 180
 - static, 257
 - symbolic, 268
- lint, 162
- listen.h header file, 162
- Literal string, 227
- Loader, 160
- Local domain, 69
- locale, date and time, 49
- Locale, 162, 165, 287
- locale.h header file, 165
- Localization
 - LANG environment variable, 151
 - date and time, 276
 - defined, 165

Localization (*continued*)
 locale, 162
 message catalog, 181
 localization, date and time, 49
 localtime(S), discussion, 276
 lockcmn.h header file, 39
 lockf(S) example, 85
 lock.h header file, 166
 locking.h header file, 39
 Lomuto, Ann Nicols, 320
 Lomuto, Nico, 320
 longjmp, 241
 Low-level language, 116
 Low-level routines, 92
 lprof, 220
 lseek(S), 210
 lseek(S) example, 85
 LUID (logged-in user identification), 237

M

m4
 macro processor, 167
 preprocessor, 213
 Macro
 defined, 167
 transform macro parameters, 262
 Macro processor, m4, 167
 Magic file in /etc, 169
 Magic number
 archive library, 9
 defined, 169
 Major number, 169
 make(CP), 170
 malloc, 170
 malloc.h header file, 171
 Manifest constants, 26, 39
 Manual pages, 171
 Mason, Tony, 320
 Math routines, random number generator,
 225
 Mathematics
 routines, 174
 typesetting, 82
 matherr(S), 174
 math.h header file, 176
 MAXSLICE tunable kernel parameter, 233
 M_COFF manifest constant, 39
 m_defs.h header file, 176
 mdevice file, 176

Medium memory model, 179
 Member, defined, 263
 Memory
 allocation routines, 177
 buffer, 21
 management, 178
 manipulation routines, 22
 models, 179
 paging, 198
 Memory Management Unit (MMU), 296
 memory.h header file, 180
 menu.h header file, 181
 Message catalog
 NLSPATH environment variable, 187
 defined, 181
 format, 176
 Message queue flush, 104
 Messages
 IPC, 138, 182
 STREAMS, 184
 Meta-characters, 227
 Miller, Chris D.F., 320
 Minor number, 169
 mkfifo(S), 210
 mknod(S), 210
 mktime(S), discussion, 276
 MMDf, 184
 mnttab.h header file, 184
 Modem handling, 64
 mon.h header file, 39, 184
 Motif, OSF/Motif
 Motif window manager (mwm), 185
 Mount table, 101
 mouse.h header file, 185
 Multibyte characters, 186
 mwm, 185
 M_XOUT manifest constant, 39

N

nan.h header file, 187
 Native language, message catalog, 181
 Native language header file, 152
 Network Services Library, 277
 nice(C) and nice(S), 233
 NINODE tunable parameter, 122
 nlist.h header file, 187
 NLPS, listen.h header file, 162
 NLSPATH environment variable, 181, 187
 nl_types.h header file, 188

Node, device, Special device file

Null pointer, 188

Null string, 228

O

Object code, data symbol, 48

Object code, 189

Object file

defined, 160

executable files, 87

memory models, 180

Object module, binary, 17

OMF (Object Module File)

debugging, 54

OMF (Object Module Format), 37, 189, 236, 305

Open Desktop, SCO, defined, 190

Open Software Foundation (OSF), 190, 257

Open Systems Interconnection (OSI), 193

open(S), internal implementation, 98

Operator, regular expression, 227

Optimization

cursor, 88

defined, 190

Option, 192

Orange Book, 236

O'Reilly, Tim, 320

OS/2 cross-development, 179

OSF/Motif, 190

OSI Reference Model, discussion, 193

P

Package, 207

Page

defined, 197

fault, 197

Page out daemon, vhand, 296

Paging, defined, 198

panel.h header file, 199

Parameters, transform into quote strings in

macros, 262

Parent process, debugging, 54

Parent process identification (PPID), 199, 215

Parsing, 156, 199, 311

passwd file, 200

passwd structure, 290

Password, encrypting with crypt(W), 73

Pathname

conversion, 100

defined, 200

process resolution, 217

pclose(S), 210

Pending signal, 250

Performance issues, macros vs. functions, 167

Permissions, 201

Perms list, 204

Permuted index, 173

Physical segments, memory models, 179
pic

preprocessor, 209, 213

Pictures, typesetting, 209

PID, process ID

Pike, Rob, 319

Pinson, Lewis, 321

Pipe

defined, 209

file descriptors, 95

pipe(S), 210

Plauger, P.J., 319

Plum, Thomas, 320

pollfd structure, 211

poll.h header file, 211

popen(S), 210

Portable C compiler, 27, 40

POSIX

XPG/3, 304

defined, 212

standard, 257

termio, compliance, 275

unistd.h header file, 289

POSIX signal handling, 251

PPID, parent process identification

Pragma, packing structures, 264

prd keyword, 204

Predefined macros, 26

preigion table, with shared memory, 246

Preprocessor

defined, 213

eqn, 82

printf(S), using in debugging, 52

Privileges, super user, 266

proc structure

buf structure, 21

defined, 213

Process

ID (PID), 219

context switch, 40

Process (continued)

current working directory, 217
 defined, 213
 environment, 95, 214
 group ID, 219
 priority, 233
 real group ID, 226
 real user ID, 226
 root directory, 217
 states, 214
 subsystem, 219
 zombie, 313

Process identification (PID), 215

Process table

entry for zombie process, 313
 proc structure, 213

Processors

80x86, 315
 binary compatibility, 17
 prof, 220

prof.h header file, 219

Profiler, lprof and prof, 220

Program

defined, 220
 invoking, 10
 variable, 293

prot.h header file, 220

Prototype, function, 107

prototypes.h header file, 220

ps(C), 32, 214

Pseudo-drivers, 56

Pure memory model, 179

pwd.h header file, 221, 290

Q

qsort, 223

queue_t, 223

Quit signal, 223

R

Radix symbol, 152

Random number functions, 175

Random number generator, 225

RARP, 273

Raw data, 30

Raw I/O, 226

Raw queue, 30

rcc(CP), 40

Read permission, 202

read(S), internal implementation, 99

Real user ID, 215, 226

Record

defined, 226

handling, 226

Record locking

defined, 93

example of, 85

regex.h header file, 227

Region, data structure, 227

Region table, with shared memory, 246

Register allocation, global, 191

Regular expression

defined, 227

header file, 227

Relative path, 201

Relocation data, 230

reloc.h header file, 230

Reverse Address Resolution Protocol, 273

Ritchie, Dennis M., 319

Rochkind, Mark, 320

Routines, random-number, 225

RTS flow control, 102

Rutter, Andy, 317

S

s5param.h header file, 20

Salt value, 73

SCCS (Source Code Control System), 232

scs.h header file, 233

Scheduler, 233

Schreiner, Axel T., 320

scnhdr.h header file, 233

SCSI

bus, 234

defined, 233

device classes, 234

sdb, using in debugging, 55

sdevice file, 235

search.h header file, 236

Section, defined, 236

Section header

COFF, 268

scnhdr.h header file, 233

Security

TCB, 272

defined, 236

trusted facilities, 280

security.h header file, 237
 Semaphores, UNIX, 138
 Server, X Windows, 299
 setjmp.h header file, 241
 setuid(S), 203
 Seven layer model, 193
 SGML, 242
 sgTTY.h header file, 242
 Shared library, 38, 243
 Shared memory
 IPC, 39, 246
 UNIX, 138
 Shared memory, 215
 sh(C), set TZ variable, 287
 sh(C), 20
 Shell
 csh, 27
 ksh, 149
 variable, 293
 Shell script
 Bourne shell, 20
 command line argument, 11
 shmat(S), 246
 shmdt(S), 246
 shmget(S), 246
 shm.h header file, 39
 shmid_ds data structure, 247
 shmop(S), 246
 sigaction structure, 252
 sigaction(S), 252
 sigaddset(S), 251
 sigdelset(S), 251
 SIG_DFL and SIG_IGN, 252
 sigemptyset(S), 251
 sigfillset(S), 251
 SIG_HOLD, 252
 sighold(S), 252
 sigignore(S), 252
 SIGKILL and SIGSTOP, 250
 Signal, blocked process, 214
 signal.h header file, 39
 Signals
 SIGCLD, 33
 quit, 223
 Signals, 249
 sigelse(S), 252
 sigsetv(S), 252
 sleep, defined, 214
 Small memory model, 179
 Sobell, Mark G., 320
 Source code, 254

space.c file, 161
 Special character, regular expressions, 227
 Special device file
 inode, 120
 Special device file, 63
 spl (set priority level), 255
 Stack, defined, 256
 Standard Generalized Markup Language, 242
 Standard I/O
 defined, 256
 low-level routines, 92
 Standards
 defined, 257
 iBCS, 122
 /usr/group, 291
 Static linking, 257
 STDC (__STDC__ Variable), 8
 stderr
 defined, 256
 example, 84
 stdin, defined, 256
 stdlib.h header file, 258
 stdout, defined, 256
 Steele, Guy L., 319
 Stevens, W. Richard, 320
 Sticky bit, 258
 storclass.h header file, 258
 STREAMS
 buffer, 21
 defined, 259
 flush a message queue, 104
 STREAMS Primer, 317
 STREAMS Programmer's Guide, 317
 Streams (Standard I/O), 260
 strftime(S), discussion, 276
 String
 definition, 261
 examples, 227
 manipulation routines, 261
 paste together, 262
 string.h header file, 262
 Stringizing, 262
 stropts.h header file, 263
 Stroustrup, Bjarne, 318, 321
 Structure
 buf, 21
 defined, 263
 tty, 283
 Structured information, 242
 stubs.c file, 161

- Subdomain, 69
- Subexpressions, 228
- Subprogram, 220
- Subsystems
 - I/O, 127
 - defined, 264
- Super block, 100-101, 264
- Super user
 - permissions, 202
 - process, 266
- Symbol
 - table, 54, 268
- Symbolic links, 268
- syms.h header file, 268
- Synchronization, clock, 34
- Syntactical elements, 311
- Syntax, 269
- System calls
 - defined, 269
 - documentation, 172
- System error handling, 83
- System V Interface Definition (SVID), 257, 269

T

- Tables, typesetting, 272
- TAM transition library, 271
- tam.h header file, 271
- Target machine, 271
- tar.h header file, 271
- tbl
 - defined, 272
 - preprocessor, 213
- tcap.h header file, 272
- TCB, 272
- /tcb/files/auth hierarchy, 280
- TCGETA and TCSETA example, 30
- TCIOFF and TCION flow controls, 102
- TCOOFF and TCOON flow controls, 102
- TCP/IP
 - XTI, 306
 - defined, 272
- TERM environment variable, 274
- termcap, 272-273
- Terminal Access Method, 271
- Terminal device queue, 30
- Terminals, block mode display processing, 274
- terminfo, 274

- termio, POSIX compliant, 275
- termio.h header file, 275
- termios.h header file, 275
- t_errno variable, 275
- .text
 - reloc.h header file, 230
 - section, 236
- Text formatting, 279
- .text section, 214
- Time
 - TZ, 276
 - defined, 276
 - epoch, 49
 - operating system, 49
 - zones, 276
- time.h header file, 277
- time(S), 276
- tinio.h header file, 39, 277
- TLB, 279
- TLI
 - XTI, 305
 - error handling, 275
 - listen.h header file, 162
 - overview, 277
- Tokens
 - lex, 156, 311
 - lexical analysis, 156
 - parsing, 199
 - syntax, 269
- Tondo, Clovis L., 321
- Translation Lookaside Buffers, 279
- Transmission Control Process, 272
- Transport Layer Interface
 - See TLI
 - XTI, 305
- Trigonometric functions, 175
- troff
 - defined, 279
 - eqn preprocessor, 82
 - pic preprocessor, 209
 - tbl preprocessor, 272
- Trusted Computing Base, 272
- Trusted facilities, 15, 220, 237, 272, 280
- tty
 - device access, 30
 - structure, 283
- tunable.h header file, 161
- Type size, C compiler, 285
- Typesetting software, 279
- types.h header file, 286
- TZ environment variable, 286

TZ variable, setting, 287
 TZ(M), 276
 tzset(S), discussion, 276

U

u_area, 291
 UART (Universal Asynchronous Receiver/Transmitter), 289
 ublock, 291
 UDP, 273
 UID, *See* User ID, 290
 Uniform Draft Standard, 291
 unistd.h header file, 289
 UNIX Operating System, 289
 unlink(S), 210
 User address space, 290
 User Datagram Protocol, 273
 User ID
 defined, 290
 effective, 87, 202
 logged-in, 237
 permissions, 203
 real, 226
 User identification, 215
 user structure, 291
 /usr/group standard, 291
 /usr/include directory, 115
 /usr/lib/nls, 181
 ustat.h header file, 291
 Utility, 291
 utime.h header file, 292
 utmp.h header file, 292
 utsname.h header file, 39

V

values.h header file, 293
 varargs.h header file, 293
 varargs(S), 107
 Variable
 defined, 293
 environ, 81
 environment, 81
 VGA (Video Graphics Array), 111
 vhand
 daemon, 198
 defined, 296
 Virtual address
 buf structure, 21

Virtual address (*continued*)
 mapping, 296

W

waitpid(S), 33
 wait(S), 33, 313
 wakeup, defined, 214
 wchar_t, 186
 Weinberger, Peter J., 317
 Weiner, Richard S., 321
 White space
 defined, 297
 with stringizing operators, 262
 Wide characters, 186
 Widget, 297
 Window, curses, 298
 Word, 298
 Write permission, 202
 write(S), internal implementation, 99
 WYSIWYG text editors, 279
 X Window System
 defined, 299
 intrinsics, 126
 X Window Systems, debugging, 54
 X Windows System
 Open Desktop, 190
 widget, 297
 xdata.h header file, 300
 XENIX cross development, 300
 XENIX cross-development, 179
 XENIX operating system, 304
 XON and XOFF flow controls, 102
 X/Open, standard, 257
 X/Open Portability Guide
 XPG/3
 X/Open Transport Interface, 305
 x.out.h header file, 305
 XPG/3, 304
 XTI
 defined, 305
 error handling, 275
 error return codes, 308
 restrictions, 308
 states, 309
 xti.h header file, 306, 309

Y

yacc, bibliographic entry, 320

yacc, 311

Young, Douglas A., 321

Z

Zombie state, 214, 313

0-9

286 binaries, debugging, 55

286 systems, stringizing operators, 262

80286 processor, memory models for, 179

8086 processor, memory models for, 179

80x86 processors, family of, 315



Please help us to write computer manuals that meet your needs by completing this form. Please post the completed form to the Technical Publications Research Coordinator nearest you: The Santa Cruz Operation, Ltd., Croxley Centre, Hatters Lane, Watford WD1 8YN, United Kingdom; The Santa Cruz Operation, Inc., 400 Encinal Street, P.O. Box 1900, Santa Cruz, California 95061, USA or SCO Canada, Inc., 130 Bloor Street West, 10th Floor, Toronto, Ontario, Canada M5S 1N5.

Volume title: _____
(Copy this from the title page of the manual, for example, SCO UNIX Operating System User's Guide)

Product: _____
(for example, SCO UNIX System V Release 3.2 Operating System Version 4.0)

How long have you used this product?

- ☐ Less than one month ☐ Less than six months ☐ Less than one year
☐ 1 to 2 years ☐ More than 2 years

How much have you read of this manual?

- ☐ Entire manual ☐ Specific chapters ☐ Used only for reference

	Agree			Disagree		
The software was fully and accurately described	☐	☐	☐	☐	☐	☐
The manual was well organized	☐	☐	☐	☐	☐	☐
The writing was at an appropriate technical level (neither too complicated nor too simple)	☐	☐	☐	☐	☐	☐
It was easy to find the information I was looking for	☐	☐	☐	☐	☐	☐
Examples were clear and easy to follow	☐	☐	☐	☐	☐	☐
Illustrations added to my understanding of the software	☐	☐	☐	☐	☐	☐
I liked the page design of the manual	☐	☐	☐	☐	☐	☐

If you have specific comments or if you have found specific inaccuracies, please report these on the back of this form or on a separate sheet of paper. In the case of inaccuracies, please list the relevant page number.

May we contact you further about how to improve SCO UNIX documentation?
If so, please supply the following details:

Name _____ Position _____

Company _____

Address _____

City & Post/Zip Code _____

Country _____

Telephone _____ Facsimile _____



10 December 1991



BH01 400P000

58085

SCO UNIX Development System Encyclopedia

CLUSTER



BJ01400P000

MANUAL



AU01404P000