

SCO

SCO OpenServerTM Development System

Developer's Topics

SCO
OpenServerTM

SCO OpenServerTM
Development System
Developer's Topics

© 1983–1995 The Santa Cruz Operation, Inc.; © 1989–1994 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 Street, Santa Cruz, California, 95060, USA. Copyright infringement is a serious matter under the United States and foreign Copyright Laws.

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.

This SCO OpenServer Development System software is commercial computer software and, together with any related documentation, is subject to the restrictions on U.S. Government use as set forth below. If this procurement is for a DOD agency, the following DFAR Restricted Rights Legend applies:

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013. Contractor/Manufacturer is The Santa Cruz Operation, Inc., 400 Encinal Street, Santa Cruz, CA 95060.

If this procurement is for a civilian government agency, the following FARS Restricted Rights Legend applies:

RESTRICTED RIGHTS LEGEND: This computer software is submitted with restricted rights under Government Contract No. ____ (and Subcontract No.____, if appropriate). It may not be used, reproduced, or disclosed by the Government except as provided in paragraph (g)(3)(i) of FAR Clause 52.227-14 alt III or as otherwise expressly stated in the contract. Contractor/Manufacturer is The Santa Cruz Operation, Inc., 400 Encinal Street, Santa Cruz, CA 95060.

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. This SCO OpenServer Development System software includes software that is protected by these copyrights:

© 1983–1995 The Santa Cruz Operation, Inc.; © 1987 Apollo Computer Inc.; © 1988–1991 Apple Computer, Inc.; © 1976–1992 AT&T; © 1986–1987 Convergent Technologies Inc.; © 1987–1992 Digital Equipment Corporation; © 1993 D.I.S - Università' di Pavia; © 1987–1988 Don Bennett; © 1988 Evans & Sutherland Computer Corporation; © 1991–1992 Fuji Xerox Co., Ltd.; © 1991–1992 Fujitsu Limited; © 1990–1993 Groupe Bull; © 1986–1992 Hewlett-Packard Company; © 1990 Ing.C.Olivetti & C., S.p.A.; © 1990 Intel Corporation; © 1990 Interactive Systems Corporation, (ISC); © 1988–1994 IXI Limited; © 1991–1994 Karl Lehenbauer and Mark Diekhans; © 1990 Larry Medwin; © 1986–1988 Larry Wall; © 1987–1994 Legent Corporation; © 1988–1993 Locus Computing Corporation; © 1985–1993 Massachusetts Institute of Technology; © 1989 Matra Datavision Inc.; © 1990 Megatek Corporation; © 1980–1989 Microsoft Corporation; © 1990 Motorola Inc.; © 1989–1991

National Computer Graphics Association; © 1990 NCR Corporation; © 1989–1993 Network Computing Devices, Inc.; © 1990–1991 NTT Software Corporation; © 1989 Object Design, Inc.; © 1990 Oki Electric Industry Co., Ltd.; © 1992 Oki Technosystems Laboratory, Inc.; © 1990–1991 OMRON Corporation; © 1989–1994 Open Software Foundation, Inc.; © 1982 Pavel Curtis; © 1989 Prentice Hall; © 1979–1994 Regents of the University of California; © 1987–1990 Siemens Nixdorf Informationssysteme AG; © 1991–1992 Sony Corporation; © 1990–1992 SORD Computer Corp.; © 1986–1992 Sun Microsystems Inc.; © 1988–1991 Tektronix Inc.; © 1990–1992 Toshiba Corp.; © 1990–1991 UniSoft Group Limited; © 1990 Unisys Corporation; © 1986 University of Toronto; © 1978–1993 UNIX System Laboratories, Inc.; © 1988 Wyse Technology, Inc.; © 1988–1992 X Consortium. All Rights Reserved. SCO NFS and SCO TCP/IP were developed by Legent Corporation based on Lachman System V NFS. SCO TCP/IP is derived from Lachman System V STREAMS TCP, a joint development of Legent Corporation and Convergent Technologies.

SCO, the SCO logo, The Santa Cruz Operation, Multiscreen, Open Desktop, ODT, SCO Global Access, SCO OpenServer, SCO MPX, SCO MultiView, SCO Visual Tcl, and VP/ix are trademarks or registered trademarks of The Santa Cruz Operation, Inc. in the USA and other countries. UNIX is a registered trademark in the USA and other countries, licensed exclusively through X/Open Company Limited. All other brand and product names are or may be trademarks of, and are used to identify products or services of, their respective owners.

Date: 31 January 1995

Document Version: 5.0

About this book	1
Typographical conventions	3
Related documentation	4
Commercial books and articles	6
How can we improve this book?	7

Chapter 1

Floating point operations 9

IEEE arithmetic	10
Single-precision	10
Double-precision	10
Extended-precision	11
Normalized numbers	11
Denormalized numbers	12
Maximum and minimum representable floating point values	12
Special-case values	12
NaNs and infinities	13
Rounding control	14
Exceptions, sticky bits, and trap bits	15
Single-precision floating point operations	16
Double-extended-precision	17
IEEE requirements	18
Conversion of floating point formats to integer	18
Square root	18
Compares and unordered condition	18
NaNs and infinities in input/output	19

Chapter 2

Complying with standard C 21

Mixing old and new style functions	22
Writing new code	22
Updating existing code	22
Mixing considerations	23
Examples	24
Functions with varying arguments	25
Example	25

Promotions: unsigned vs. value preserving	28
Background	28
Compilation behavior	28
First example: using a cast	29
Bit-fields	30
Second example: result is the same	30
Integral constants	30
Third example: integral constants	31
Tokenization and preprocessing	32
ANSI C translation phases	32
Old C translation phases	33
Logical source lines	33
Macro replacement	34
Stringizing	34
Token pasting	35
Using const and volatile	36
Types for lvalues	36
Type qualifiers in derived types	36
Using const to read character values	37
When to use volatile	38
Multibyte characters and wide characters	39
“Asianization” means multibyte characters	39
Encoding variations	40
Wide characters	40
Conversion functions	40
C language features	41
Standard headers and reserved names	42
Balancing process	42
Standard headers	43
Names reserved for implementation use	43
Names reserved for expansion	44
Names safe to use	45
Internationalization	45
Locales	45
The setlocale() function	46
Changed functions	47
New functions	47
Grouping and evaluation in expressions	48
Definitions	48
The Kernighan and Ritchie C rearrangement license	49
The ANSI C rules	50

Parentheses grouping and evaluation	50
The “as if” rule	50
Incomplete types	51
Types	51
Completing incomplete types	51
Declarations	52
Expressions	52
Justification	53
Examples	53
Compatible and composite types	54
Multiple declarations	54
Separate compilation compatibility	54
Single compilation compatibility	54
Compatible pointer types	55
Compatible array types	55
Compatible function types	55
Special cases	56
Composite type	56

Chapter 3

Constructing and using message catalogs 57

Creating a message catalog	58
Code example using a message catalog	59
Combining the message catalog and the application program	61
Running the application program	62

Chapter 4

Using trusted facilities 63

Using protected subsystems	64
Guidelines for using trusted facilities	64
Programming examples	65
subsys.c example	66
prwarn.c example	71
loge.c example	89

Chapter 5

Using the event manager API 113

Operation and semantics overview	114
Event generating hardware	115
Using the event queue	115
Event manager calls	116
A sample program	117

Chapter 6

Job control under ksh 125

Chapter 7

Common Object File Format (COFF) 131

Definitions and conventions	133
Sections	133
Physical and virtual addresses	133
Target machine	133
File header	134
Magic numbers	134
Flags	134
File header declaration	135
Optional header information	136
Standard UNIX system <i>a.out</i> header	136
Optional header declaration	137
Section headers	138
Flags	139
Section header declaration	140
.bss section header	140
Sections	141
Relocation information	141
Relocation entry declaration	142
Line numbers	143
Line number declaration	144
Symbol table	144
Special symbols	146

Inner blocks	147
Symbols and functions	148
Symbol table entries	148
Symbol names	149
Storage classes	150
Storage classes for special symbols	151
Symbol value field	152
Section number field	153
Section numbers and storage classes	154
Type entry	155
Type entries and storage classes	157
Structure for symbol table entries	158
Auxiliary table entries	159
File names	160
Sections	160
Tag names	161
End of structures	161
Functions	162
Arrays	162
End of blocks and functions	163
Beginning of blocks and functions	163
Names related to structures, unions, and enumerations	164
Auxiliary entry declaration	164
String table	166
Access routines	166

Chapter 8

ELF object files 169

File format	169
Data representation	170
Program linking	171
Section header	177
String table	187
Symbol table	187
Relocation	192
Program execution	194
Program header	194
Program loading (processor-specific)	201
Program interpreter	204

Dynamic linker	205
----------------------	-----

Chapter 9

Enhanced asm facility 215

Definition of terms	216
Example	216
Definition	217
Use	217
Using asm macros	218
Definition	218
Writing asm macros	220

Chapter 10

Mapfile option 223

Using the mapfile option	224
Mapfile structure and syntax	224
Segment declarations	225
Mapping directives	227
Size-Symbol declarations	229
Mapping example	229
Mapfile option defaults	231
Internal map structure	233
Error messages	235
Warnings	236
Fatal errors	236

Chapter 11

Integrating applications into the Desktop 239

What you should know	240
Graphical interface standards and libraries	241
Backward compatibility	241
Putting your icon on the Desktop	242
Create your icon pixmaps	243
Define icon triggers	246
Install icon files	250

Animating icons	252
Localizing icon labels and messages	253
Creating icons for user data files	255
Adapting to display resolution at run time	257
Supporting the Desktop color selector	261
Defining display-dependent colors	264
Communicating with the session manager	266

Chapter 12

Custom Distribution Mastering Toolkit 271

Creating custom-installable software distributions	272
About Software Storage Objects (SSO)	273
Planning your product structure	275
Determining SSO file attributes	276
Setting up the distribution tree	277
Setting up the CDMT environment	278
Creating CDMT input files	278
Creating and populating the SSO tree	282
Creating custom-installable SSO archives	283
Advanced software installations	285
About component scripts	285
Software management phases	286
Creating software upgrades	298
About the SSO Path Mapping API	301
Using the SSO Path Mapping API	302

Bibliography 307

Books	307
Periodicals	315
SCO resources	316

Index 319

About this book

Developer's Topics contains technical papers on topics relevant to programmers. Many of these papers include extensive examples that illustrate SCO® features added to the porting base. These topics are not well-covered in most industry-standard books, and require more information than can be incorporated in the *Programmer's Reference Manual*.

The papers included are:

Chapter 1, "Floating point operations" (page 9) discusses hardware and software issues associated with floating point operations.

Chapter 2, "Complying with standard C" (page 21) discusses proper coding methodologies associated with producing ANSI-compliant C code.

Chapter 3, "Constructing and using message catalogs" (page 57) discusses how to internationalize software using message catalogs. Included is an annotated example of constructing and using a simple message table within code.

Chapter 4, "Using trusted facilities" (page 63) provides three annotated examples of applications coded to run under the trusted facility software.

Chapter 5, "Using the event manager API" (page 113) discusses how to use the event manager API to handle devices such as mice within curses-based applications.

Chapter 6, "Job control under ksh" (page 125) discusses how to implement job control under the Korn Shell. This chapter includes an annotated example of a **curses(S)** application coded for job control.

Chapter 7, “Common Object File Format (COFF)” (page 131) defines the Common Object File Format and highlights issues associated with COFF binaries.

Chapter 8, “ELF object files” (page 169) defines the Extended Linker Format (ELF) and highlights issues associated with ELF binaries.

Chapter 9, “Enhanced asm facility” (page 215) discusses the enhanced **asm** facility available with the SCO OpenServer™ Development System.

Chapter 10, “Mapfile option” (page 223) discusses mapfiles used by the linker to control memory allocation.

Chapter 11, “Integrating applications into the Desktop” (page 239) describes how to integrate applications into the Open Desktop environment.

Chapter 12, “Custom Distribution Mastering Toolkit” (page 271) discusses use of the Custom Distribution Mastering Toolkit (CDMT) and how to create software installable with the **custom** utility.

You will find the information you need more quickly if you are familiar with:

- the typographical conventions used in our books (page 3)
- what other documentation is available (page 4)

Although we try to present information in the most useful way, you are the ultimate judge of how well we succeed. Please let us know how we can improve this book (page 7).

Typographical conventions

This publication presents commands, filenames, keystrokes, and other special elements in these typefaces:

Example:	Used for:
lp or lp(C)	commands, device drivers, programs, and utilities (names, icons, or windows); the letter in parentheses indicates the reference manual section in which the command, driver, program, or utility is documented
<i>/new/client.list</i>	files, directories, and desktops (names, icons, or windows)
<i>root</i>	system, network, or user names
<i>filename</i>	placeholders (replace with appropriate name or value)
⟨Esc⟩	keyboard keys
Exit program?	system output (prompts, messages)
yes or yes	user input
"Description"	field names or column headings (on screen or in database)
Cancel	button names
Edit	menu names
Copy	menu items
File ⇌ Find ⇌ Text	sequences of menus and menu items
open or open(S)	library routines, system calls, kernel functions, C keywords; the letter in parentheses indicates the reference manual section in which the file is documented
\$HOME	environment or shell variables
SIGHUP	named constants or signals
"adm3a"	data values
<i>employees</i>	database names
<i>orders</i>	database tables
buf	C program structures
<i>b_b.errno</i>	structure members

Related documentation

The SCO OpenServer™ Development System includes comprehensive documentation. Books are available in online and/or printed form. Access online books by double-clicking on the Desktop **Help** icon. Printed versions of the books are also available.

The contents of the SCO OpenServer Development System documentation set are illustrated here. In addition to these books, a copy of the *Development System Release and Installation Notes* is provided with the Development System software.

	Reference	Guide
UNIX® Applications	• C++ Library Manual • C++ Product Reference Manual • Programmer's Reference Manual (2 Volumes)	• C++ Selected Readings • Character User Interfaces Guide • Developer's Topics • Programming Tools Guide
Networking	• Network Programmer's Guide and Reference	
Graphical	• X Window System Programmer's Reference • OSF/Motif Programmer's Reference	
Optional	• Device Driver Writer's Guide	

The books included with the Development System are:

C++ Library Manual

describes the three C++ class libraries and their use.

C++ Product Reference Manual

provides a complete definition of the C++ language supported by Release 3.1 of the SCO® C++ Language System.

C++ Selected Readings

contains papers describing aspects of the C++ language.

Character User Interfaces Guide

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

Developer's Topics

a collection of technical papers about topics of interest to users of the Development System. Many of these papers include examples illustrating features that are extensions provided by SCO software.

Network Programmer's Guide and Reference

describes the libraries and application program interfaces (APIs) available (Internet domain sockets, TLI, XTI, ONC, RPC, and XDR), for developing applications that use the services of network-oriented communication protocols (TCP/IP, IPX/SPX, and NetBIOS). It also describes how to use UNIX domain sockets for interprocess communication, and an SMUX API to create SMUX peers for SNMP agents. The (NC), (NET), (NS), (SAPD), (SLIB), and (SSC) reference manual pages are included in this book.

OSF/Motif Programmer's Reference

consists of the OSF/Motif toolkit, window manager, and user interface language commands and functions: (Xm) reference manual pages.

Programmer's Reference Manual

a two-volume set of reference manual pages for the Development System, including sections (CDMT), (CP), (FP), and (S).

Programming Tools Guide

provides useful information about programming tools and their use in developing software, including implementation-specific details about the ANSI-conforming C compiler provided with the Development System. It also contains information about debugging tools and their use in tracking down and eliminating problems in C and assembly language programs, and information on using **dbXtra**[™] to debug C and C++ programs in an OSF/Motif[™] based windowing environment.

Release and Installation Notes

highlights new features, product changes, documentation changes, and known limitations for this release of the Development System. It also contains the installation notes.

X Window System Programmer's Reference

provides X Window System[™] programming commands and functions: X Library (XS), X Toolkit (Xt), X Miscellaneous Utilities (Xmu), and X Extensions (Xext) reference manual pages for programmers.

In addition, the following book is available on request:

Device Driver Writer's Guide

provides guide and reference information about writing device drivers for SCO systems. This book is available separately.

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. The *Developer's Topics* includes a large "Bibliography" that will interest users of the Development System.

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

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

One of the following general books about programming on UNIX operating systems:

Donald Lewine, *POSIX Programmer's Guide*.

Marc Rochkind, *Advanced UNIX Programming*.

W. Richard Stevens, *Advanced Programming in the UNIX Environment*.

W. Richard Stevens, *UNIX Network Programming*. Useful for programmers who are writing distributed applications.

Maurice Bach, *Design of the UNIX Operating System*. Useful for programmers who want to thoroughly understand UNIX system internals.

See the "Bibliography" for full citations if you are not familiar with the books listed above.

How can we improve this book?

What did you find particularly helpful in this book? Are there mistakes in this book? Could it be organized more usefully? Did we leave out information you need or include unnecessary material? If so, please tell us.

To help us implement your suggestions, include relevant details, such as book title, section name, page number, and system component. We would appreciate information on how to contact you in case we need additional explanation.

To contact us, use the card at the back of the *SCO OpenServer Handbook* or write to us at:

Technical Publications
Attn: CFT
The Santa Cruz Operation, Inc.
PO Box 1900
Santa Cruz, California 95061-9969
USA

or e-mail us at:

techpubs@sco.com or ... *uunet!sco!techpubs*

Thank you.

Chapter 1

Floating point operations

The C compiler in the *Programmer's Reference Manual Volume 1* uses the IEEE standard single-precision, double-precision and extended-precision data types, operations, and conversions. Library functions are provided for further IEEE support.

You will probably not need any special functions to use floating point operations in your programs. If you do, however, you can find information about floating point support in this chapter.

NOTE For more information on how the C compilation system supports the IEEE standard see "IEEE requirements" (page 18).

This chapter contains sections on the following topics:

- the details of IEEE arithmetic
- floating point exception handling
- conversion between binary and decimal values
- single-precision floating point operations
- implicit precision of subexpressions
- IEEE Requirements

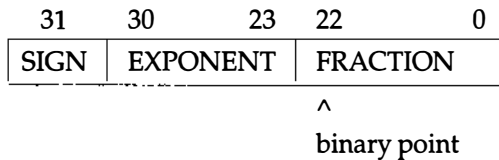
IEEE arithmetic

This section provides the details of floating point representation, the environment, and exception handling. Most users need not be concerned with the details of the floating point environment.

NOTE The floating point subsystems of the Intel386™ microprocessor are based on the Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985. For more information about this standard, write to IEEE Service Center, 445 Hoes Lane, Piscataway, NJ, 08854, or call (201) 981-0060

Single-precision

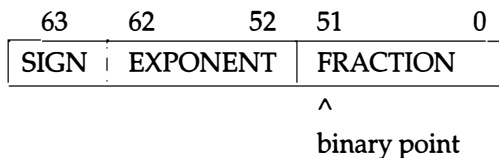
Single-precision floating point numbers have the following format:



Field	Position	Full name
sign	31	sign bit (0 == positive, 1 == negative)
exponent	30-23	exponent (biased by 127)
fraction	22-0	fraction (bits to right of binary point)

Double-precision

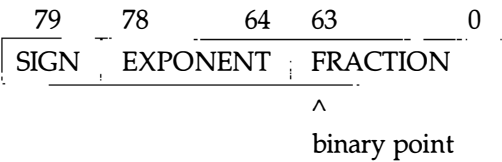
Double-precision floating point numbers have the following format:



Field	Position	Full name
sign	63	sign bit (0==positive, 1==negative)
exponent	62-52	exponent (biased by 1023)
fraction	51-0	fraction (bits to right of binary point)

Extended-precision

Extended-precision floating point numbers have the following format on the Intel386™ microprocessor:



For extended precision, bit 63 is always a 1, and the decimal point comes after this bit. The other formats have an implicit bit 1 before the fraction, as explained in "Normalized numbers" (this page).

Field	Position	Full name
sign	79	sign bit (0==positive, 1==negative)
exponent	78-64	exponent (biased by 16,383)
fraction	63-0	fraction (bit 63=1, followed by the decimal point. Bits 62-0 are the fraction.

Normalized numbers

A number is normalized if the exponent field contains other than all 1's or all 0's.

The exponent field contains a biased exponent, where the bias is 127 in single-precision, 1023 in double-precision, and 16,383 in extended-precision. Thus, the exponent of a normalized floating point number is in the range -126 to 127 inclusive for single-precision, and in the range -1022 to 1023 inclusive for double-precision. For extended-precision the range is -16,382 to 16,383.

There is an implicit bit associated with both single- and double-precision formats. The implicit bit is not explicitly stored anywhere (thus its name). Logically, for normalized operands the implicit bit has a value of 1 and resides immediately to the left of the binary point (in the 2⁰ position). Thus the implicit bit and fraction field together can represent values in the range 1 to 2-2⁻²³ inclusive for single-precision, and in the range 1 to 2-2⁻⁵² inclusive for double-precision. For extended-precision, there is no such bit, therefore the field can represent values in the range 1 to 2-2⁻⁶³.

Thus normalized single-precision numbers can be in the range (plus or minus) 2⁻¹²⁶ to (2-2⁻²³) × 2¹²⁷ inclusive.

Normalized double-precision numbers can be in the range (plus or minus) 2^{-1022} to $(2-2^{-52}) \times 2^{1023}$ inclusive.

Normalized extended-precision numbers can be in the range (plus or minus) $2^{-16,382}$ to $(2-2^{-63}) \times 2^{16,383}$ inclusive.

Denormalized numbers

A number is denormalized if the exponent field contains all 0's and the fraction field does not contain all 0's.

Thus denormalized single-precision numbers can be in the range (plus or minus) $2^{-126} \times 2^{-22} = 2^{-148}$ to $(1-2^{-22}) \times 2^{-126}$ inclusive.

Denormalized double-precision numbers can be in the range (plus or minus) $2^{-1022} \times 2^{-51} = 2^{-1073}$ to $(1-2^{-51}) \times 2^{-1022}$ inclusive.

Denormalized extended-precision numbers do not have a 1 bit in position 63. Therefore, it stores numbers in the range (plus or minus) $2^{-16,382} \times 2^{-63} = 2^{-16,445}$ to $(1-2^{-63}) \times 2^{-16,382}$ inclusive.

Both positive and negative zero values exist, but they are treated the same during floating point calculations.

Maximum and minimum representable floating point values

The maximum and minimum representable values in floating point format are defined in the header file *values.h*.

Special-case values

The following table gives the names of special cases and how each is represented.

Value name	Sign	Exponent	Fraction	
			MSB	Rest of fraction
NaN (non-trapping)	X	Max	0	Nonzero
Trapping NaN	X	Max	1	X
Positive Infinity	0	Max	Min	
Negative Infinity	1	Max	Min	
Positive Zero	0	Min	Min	
Negative Zero	1	Min	Min	
Denormalized number	X	Min	Nonzero	
Normalized number	X	NotMM	X	

Key:

X	does not matter
Max	maximum value that can be stored in the field (all 1's)
Min	minimum value that can be stored in the field (all 0's)
NaN	not a number
NotMM	field is not equal to either Min or Max values
Nonzero	field contains at least one "1" bit
MSB	Most Significant Bit

The algorithm for classification of a value into special cases follows:

```

If (Exponent==Max)
  If (Fraction==Min)
    Then the number is Infinity (Positive or Negative
      as determined by the Sign bit).
  Else the number is NaN (Trapping if FractionMSB==0,
    non-Trapping if FractionMSB==1).

Else If (Exponent==Min)
  If (Fraction==Min)
    Then the number is Zero (Positive or Negative
      as determined by the Sign bit).
  Else the number is Denormalized.
Else the number is Normalized.

```

NaNs and infinities

The floating point system supports two special representations:

Infinity	Positive infinity in a format compares greater than all other representable numbers in the same format. Arithmetic operations on infinities are quite intuitive. For example, adding any representable number to infinity is a valid operation the result of which is positive infinity. Subtracting positive infinity from itself is invalid. If some arithmetic operation overflows, and the overflow trap is disabled, in some rounding modes the result is infinity.
Not-a-Number (NaN)	These floating point representations are not numbers. They can be used to carry diagnostic information. There are two kinds of NaNs: signaling NaNs and quiet NaNs. Signaling NaNs raise the invalid operation exception whenever they are used as operands in floating point operations. Quiet NaNs

propagate through most operations without raising any exception. The result of these operations is the same quiet NaN. NaNs are sometimes produced by the arithmetic operations themselves. For example, 0.0 divided by 0.0, when the invalid operation trap is disabled, produces a quiet NaN.

The header file *ieeefp.h* defines the interface for the floating point exception and environment control. This header defines three interfaces:

- Rounding control
- Exception control
- Exception handling

Rounding control

The floating point arithmetic provides four rounding modes that affect the result of most floating point operations. (These modes are defined in the header *ieeefp.h*):

```
FP_RN  round to nearest representable number, tie -> even
FP_RP  round toward plus infinity
FP_RM  round toward minus infinity
FP_RZ  round toward zero (truncate)
```

You can check the current rounding mode with the function

```
fp_rnd fpgetround(void); /* return current rounding mode */
```

You can change the rounding mode for floating point operations with the function:

```
fp_rnd fpsetround(fp_rnd); /* set rounding mode, */
                          /* return previous    */
```

(**fp_rnd** is an enumeration type with the enumeration constants listed and described above. The values for these constants are in *ieeefp.h*.)

NOTE The examples in this section, such as the one directly above, illustrate function prototypes. For further information on function prototypes, see “Function definitions” in the *Programming Tools Guide*.

The default rounding mode is round-to-nearest. In C, floating point to integer conversions are always done by truncation, and the current rounding mode has no effect on these operations.

For more information on **fpgetround** and **fpsetround**, see **fpgetround(S)**.

Exceptions, sticky bits, and trap bits

Floating point operations can lead to any of the following types of floating point exceptions:

- Divide by zero exception

This exception happens when a non-zero number is divided by floating point zero.

- Invalid operation exception

All operations on signaling NaNs raise an invalid operation exception. Zero divided by zero, infinity subtracted from infinity, infinity divided by infinity all raise this exception. When a quiet NaN is compared with the greater or lesser relational operators, an invalid operation exception is raised.

- Overflow exception

This exception occurs when the result of any floating point operation is too large in magnitude to fit in the intended destination.

- Underflow exception

When the underflow trap is enabled, an underflow exception is signaled when the result of some operation is a very tiny non-zero number that may cause some other exception later (such as overflow upon division). When the underflow trap is disabled, an underflow exception occurs only when both the result is very tiny (as explained above) and a loss of accuracy is detected.

- Inexact or imprecise exception

This exception is signaled if the rounded result of an operation is not identical to the infinitely precise result. Inexact exceptions are quite common. $1.0 / 3.0$ is an inexact operation. Inexact exceptions also occur when the operation overflows without an overflow trap.

NOTE The above examples for the exception types do not constitute an exhaustive list of the conditions when an exception can occur. The floating point implementation on the Intel™ processors includes another exception type called “Denormalization exception”. This exception occurs when the result of an expression is a denormalized number.

Single-precision floating point operations

The ANSI standard for C has a provision that allows expressions to be evaluated in single-precision arithmetic if there is no **double** (or **long double**) operand in the expression. The C compiler supports this provision.

Floating point constants are double-precision, unless explicitly stated to be **float**. For example, in the statements

```
float a,b;
...
a = b + 1.0;
```

because the constant **1.0** has type **double**, *b* is promoted to **double** before the addition and the result is converted back to **float**. However, the constant can be made explicitly a **float**:

```
a = b + 1.0f;
```

or

```
a = b + (float) 1.0;
```

In this case, the statement can potentially be compiled to a single instruction. Single-precision operations tend to be faster than double-precision operations.

Whether a computation can be done in single-precision is decided based on the operands of each operator. Consider the following:

```
float s;
double d;

d = d + s * s;
```

*s * s* is computed to produce a single-precision result, which is promoted to double-precision and added to *d*. Note that using single-precision (as versus double-precision) arithmetic can result in loss of precision, as illustrated in the following example.

```
float f = 8191.f * 8191.f; /* evaluate as a float */
double d = 8191. * 8191. ; /* evaluate as a double */
printf ("As float:  %f\nAs double: %f\n", f, d);
```

The result is:

```
As float: 67092480.000000
As double: 67092481.000000
```

Also, **long int** variables (same as **int**) have more precision than **float** variables. Consider the following example:

```
int i,j;
i = 0x7fffffff;
j = i * 1.0;
printf("j = %x\n", j);
j = i * 1.0f;
printf("j = %x\n", j);
```

The first **printf()** statement outputs `7fffffff`, while the second prints `0`. The second **printf()** prints `0` because the nearest **float** to `0x7fffffff` has a value of `0x80000000`. When the value is converted to an integer, the result is `0`, and a floating point imprecise result exception occurs. A trap occurs if this exception was enabled.

A function that is declared to return a **float** may actually return either a **float** or a **double**. If the function declaration is a prototype declaration in which at least one of the parameters is **float**, the function returns a **float**. Otherwise, it returns a **double** with precision limited to that of a **float**. (All of this is transparent.) For example:

```
float retflt(float);      /* actually returns a float */
float retdbl1();         /* actually returns a double */
float retdbl2(int);       /* actually returns a double */
```

Arguments work as follows:

```
double takeflt(float x);  /* takes a float */

double takedbl(x)
float x;                  /* takes a double */
```

Double-extended-precision

The C compiler in the *Programmer's Reference Manual Volume 1* produces code that uses IEEE double-extended-precision arithmetic. On implementations that do not produce IEEE double-extended-precision arithmetic, either for intermediate or final results, all results are computed with the precision implicit in their type.

ANSI C includes a new data type called **long double**, which maps to the IEEE extended-precision format. Extended-precision is a wider type than **double**. **Doubles** on the Intel386™ microprocessor are 64 bits, **long doubles** are 80 bits. All arithmetic operations (+, -, *, /) work analogously. However, ANSI C does not require a **long double** to be wider than a **double**. On Intel™ processors, complete support for **long double** is available.

IEEE requirements

All arithmetic computations generated by the C compiler strictly conform to IEEE requirements. The following is a discussion of some topics where the C compilation system falls short of completely meeting the ANSI/IEEE Standard 754-1985 requirements or the spirit of the requirements.

Conversion of floating point formats to integer

IEEE requires floating point to integer format conversions to be affected by the current rounding mode. However, the C language requires these conversions to be done by truncation (which is the same as round-to-zero). In the C compilation system floating point to integer conversions are done by truncation.

Conversion of floating point numbers to integers should signal integer overflow or invalid operation for an overflow condition. In the current implementation the integer overflow flag is set, but there is no way to enable the overflow trap. Enabling the integer overflow trap would result in a substantial performance penalty due to stalled pipeline effects.

Square root

IEEE requires the square root of a negative non-zero number to raise invalid operation, whereas UNIX operating system compatibility requires square root to return 0.0 with **errno** set to **EDOM**.

When you use the **-a ods30** option to the **cc** command, the **sqrt()** routine in the C compilation system returns 0.0 for negative non-zero inputs and sets **errno** to **EDOM**. Otherwise, the square root of a negative non-zero number raises invalid operation and returns a NaN, in strict conformance with the IEEE standard.

Compares and unordered condition

In addition to the usual relationships between floating point values (less than, equal, greater than), there is a fourth relationship: unordered. The unordered case arises when at least one operand is a NaN. Every NaN compares unordered with any value, including itself.

The C compilation system provides the following predicates required by IEEE between floating point operands:

<code>==</code>	<code>>=</code>
<code>!=</code>	<code><</code>
<code>></code>	<code><=</code>

While there is no predicate to test for unordered, you can use **isnan()** or **isnanf()** to test whether an argument is a NaN. For information on **isnan()** and **isnanf()**, see **isnan(S)**.

The relations `>`, `>=`, `<`, and `<=` raise invalid operation for unordered operands. The compiler generated code does not guard against the unordered outcome of a comparison. If the trap is masked, the path taken for unordered conditions is the same as if the conditional were true, which may result in incorrect behavior.

For the predicates `==` and `!=`, unordered condition does not lead to invalid operation. The path taken for unordered condition is the same as when the operands are non-equal, which is correct.

$(a > b)$ is not the same as $(!(a \leq b))$ in IEEE floating point arithmetic. The difference occurs when b or a compares unordered. The C compiler generates the same code for both cases.

NaNs and infinities in input/output

The **printf()** family of functions prints NaNs or infinities. NaNs are printed with their diagnostic values.

Ideally, whatever **printf()** outputs, **scanf()** should be able to read using the same format. However, **scanf()** does not recognize NaNs and infinities for floating point formats. However, since these special cases serve mostly as diagnostics for erroneous floating point computation, outputting these cases was considered more important than being able to read them.

Chapter 2

Complying with standard C

This chapter describes techniques for writing new and upgrading existing C code to comply with the ANSI C language specification.

NOTE The information in this chapter is taken from a series of articles, each covering a specific transition topic. These articles were originally written for an in-house USL newsletter by David Prosser, Distinguished Member of Technical Staff, USL.

This chapter contains the following transition topics:

- “Mixing old and new style functions” (page 22)
- “Functions with varying arguments” (page 25)
- “Promotions: unsigned vs. value preserving” (page 28)
- “Tokenization and preprocessing” (page 32)
- “Using const and volatile” (page 36)
- “Multibyte characters and wide characters” (page 39)
- “Standard headers and reserved names” (page 42)
- “Internationalization” (page 45)
- “Grouping and evaluation in expressions” (page 48)
- “Incomplete types” (page 51)
- “Compatible and composite types” (page 54)

Mixing old and new style functions

ANSI C's most sweeping change to the language is the function prototype borrowed from the C++ language. By specifying for each function the number and types of its parameters, not only does every regular compile get the benefits of **lint**-like argument/parameter checks for each function call, but arguments are automatically converted (just as with an assignment) to the type expected by the function. ANSI C includes rules that govern the mixing of old- and new-style function declarations since there are numerous lines of existing C code that could and should be converted to use prototypes.

Writing new code

When you write an entirely new program, you should use new-style function declarations (function prototypes) in headers and new-style function declarations and definitions in other C source files. However, if there is a possibility that someone will port the code to a machine with a pre-ANSI C compiler, use the macro **__STDC__** (which is defined only for ANSI C compilation systems) in both header and source files.

Because an ANSI C conforming compiler must issue a diagnostic whenever two incompatible declarations for the same object or function are in the same scope, if all functions are declared and defined with prototypes (and the appropriate headers are included by the correct source files), all calls should agree with the definition of the functions — thus eliminating one of the most common C programming mistakes.

Updating existing code

If you have an existing application and want the benefits of function prototypes, there are several possibilities for updating, depending on how much of the code you care to change:

1. Recompile without making any changes.

Even with no coding changes, the compiler will warn you about mismatches in parameter type and number when invoked with the **-v** option.

2. Add function prototypes just to the headers.

All calls to global functions are covered.

3. Add function prototypes to the headers and start each source file with function prototypes for its local (static) functions.

All calls to functions are covered, but this requires typing the interface for each local function twice in the source file.

4. Change all function declarations and definitions to use function prototypes.

For most programmers, choices 2 and 3 are probably the best cost/benefit compromise. Unfortunately, these options are precisely the ones that require detailed knowledge of the rules for mixing of old and new styles.

Mixing considerations

In order for function prototype declarations to work with old-style function definitions, both must specify functionally identical interfaces (or have “compatible types” using ANSI C’s terminology).

For functions with varying arguments, there can be no mixing of ANSI C’s ellipsis notation and the old-style `varargs()` function definition. For functions with a fixed amount of parameters, however, you can simply specify the types of the parameters as they were passed in previous implementations.

In pre-ANSI C compilers, each argument was converted just before it was passed to the called function according to the default argument promotions. These specified that all integral types narrower than `int` were promoted to `int` size, and any `float` argument was promoted to `double`. This simplified both the compiler and libraries. Function prototypes are more expressive — the specified parameter type is what is passed to the function. Thus, if a function prototype is written for an existing (old-style) function definition, there should be no parameters in the function prototype with any of the following types:

<code>char</code>	<code>signed char</code>	<code>unsigned char</code>	<code>float</code>
<code>short</code>	<code>signed short</code>	<code>unsigned short</code>	

There still remain two complications with writing prototypes: **typedef** names and the promotion rules for narrow unsigned types.

If parameters in old-style functions were declared using **typedef** names such as `off_t` and `ino_t`, it is important to know whether or not the **typedef** name designates a type that is affected by the default argument promotions. For these two, `off_t` is a **long**, so it is appropriate to use in a function prototype, but `ino_t` used to be an **unsigned** so if it were used in a prototype, the compiler would issue a diagnostic (possibly fatal) because the old-style definition and the prototype specify different and incompatible interfaces.

Just what should be used instead of an **unsigned short** leads us into the final complication. The one biggest incompatibility between pre-ANSI C compilers and ANSI C is the promotion rule for the widening of **unsigned char** and **unsigned short** to an **int**-sized value. (See “Promotions: unsigned vs. value preserving” (page 28)). Unfortunately, the parameter type that matches such an old-style parameter depends on the compilation mode used when you compile: for **-Xt**, **unsigned int** should be used; **-Xa** and **-Xc** should use **int**. The best approach (even though it violates the spirit of choices 2 and 3 above) is to change the old-style definition to specify either **int** or **unsigned int** and use the matching type in the function prototype (you can always assign its value to a local variable with the narrower type, if necessary, after you enter the function).

Examples

Appropriate use of `__STDC__` produces a header file that can be used for both old and new compilers:

```
header.h:

struct s { /* ... */ };

#ifdef __STDC__
    void errmsg(int, ...);
    struct s *f(const char *);
    int g(void);
#else
    void errmsg();
    struct s *f();
    int g();
#endif
```

The following function uses prototypes and can still be compiled on an older system:

```
struct s *

#ifdef __STDC__
f(const char *p)
#else
f(p) char *p;
#endif
{
    /* ... */
}
```

The following is an updated source file (as with choice 3 above). The local function still uses an old style definition, but a prototype is included for newer compilers:

```
source.c:

#include <header.h>
typedef /* ... */ MyType;
#ifdef __STDC__
    static void del(MyType *);
    /* ... */
#endif
static void
del(p)
    MyType *p;
{
    /* ... */
}
/* ... */
```

Functions with varying arguments

In previous implementations, you could not specify the parameter types that a function expected, but ANSI C allows you to use prototypes to do this. In order to support functions such as **printf()**, the syntax for prototypes includes a special ellipsis (...) terminator. Because an implementation might be required to do unusual things to handle a varying number of arguments, ANSI C requires that all declarations and the definition of such a function include the ellipsis terminator.

Since there are no names for the “...” part of the parameters, a special set of macros contained in *stdarg.h* gives the function access to these arguments. Earlier versions of such functions had to use similar macros contained in *varargs.h*.

Example

This example writes an error handler function called **errmsg()** that returns **void**, and whose only fixed parameter is an **int** that specifies details about the error message. This parameter may be followed by a file name, or a line number, or both, and these are followed by **printf()**-like format and arguments that specify the text of the error message.

In order to allow this example to compile with earlier compilers, it makes extensive use of the macro `__STDC__` which is defined only for ANSI C compilation systems. Thus the function's declaration (in the appropriate header file) would be:

```
#ifdef __STDC__
    void errmsg(int code, ...);
#else
    void errmsg();
#endif
```

The file that contains the definition of `errmsg()` is where the old and new styles can get complex. First, the header to include depends on the compilation system:

```
#ifdef __STDC__
#include <stdarg.h>
#else
#include <varargs.h>
#endif
#include <stdio.h>
```

(*stdio.h* is included because it is needed to call `fprintf()` and `vfprintf()` later.)

Next comes the function's definition. The identifiers `va_alist` and `va_dcl` are part of the old-style *varargs.h* interface.

```
void
#ifdef __STDC__
errmsg(int code, ...)
#else
errmsg(va_alist) va_dcl /* note: no semicolon! */
#endif
{
    /* more detail below **
}
```

Since the old-style variable argument mechanism did not allow specification of any fixed parameters (at least not officially), we need to arrange for them to be accessed before the varying portion. Also due to the lack of a name for the "..." part of the parameters, the new `va_start()` macro has a second argument — the name of the parameter that comes just before the "..." terminator.

NOTE ANSI C, as an extension, allows functions to be declared and defined with no fixed parameters, as in:

```
int f(...);
```

For such functions, **va_start()** should be invoked with an empty second argument, as in:

```
va_start(ap,)
```

The following is the body of the function:

```
{
    va_list ap;
    char *fmt;

#ifdef __STDC__
    va_start(ap, code);
#else
    int code;

    va_start(ap);
    /* extract the fixed argument */
    code = va_arg(ap, int);
#endif
    if (code & FILENAME)
        (void)fprintf(stderr, "\"%s\": ", va_arg(ap, char *));
    if (code & LINENUMBER)
        (void)fprintf(stderr, "%d: ", va_arg(ap, int));
    if (code & WARNING)
        (void)fputs("warning: ", stderr);
    fmt = va_arg(ap, char *);
    (void)vfprintf(stderr, fmt, ap);
    va_end(ap);
}
```

Both the **va_arg()** and **va_end()** macros work the same for the old-style and ANSI C versions. Because **va_arg()** changes the value of **ap**, the call to **vfprintf()** cannot be:

```
(void)vfprintf(stderr, va_arg(ap, char *), ap);
```

The definitions for the macros **FILENAME**, **LINENUMBER**, and **WARNING** are presumably contained in the same header as the declaration of **errmsg()**.

A sample call to **errmsg()** could be:

```
errmsg(FILENAME, "<command line>", "cannot open: %s\n",
       argv[optind]);
```


Promotions: unsigned vs. value preserving

The following information appears in the Rationale that accompanies the draft C Standard:

QUIET CHANGE

A program that depends on unsigned preserving arithmetic conversions will behave differently, probably without complaint. This is considered to be the most serious change made by the Committee to a widespread current practice.

This section explores how this change affects our code.

Background

According to *The C Programming Language*, Kernighan and Ritchie, (First Edition), **unsigned** specified exactly one type; there were no **unsigned chars**, **unsigned shorts**, or **unsigned longs**, but most C compilers added these very soon thereafter. (Some compilers did not implement **unsigned long** but included the other two.) Naturally, implementations chose different rules for type promotions when these new types mixed with others in expressions.

The ANSI C compiler and most other C compilers used the simpler rule — *unsigned preserving*. When an unsigned type needs to be widened, it is widened to an unsigned type; when an unsigned type mixes with a signed type, the result is an unsigned type.

The other rule, specified by ANSI C, came to be called *value preserving*, in which the result type depends on the relative sizes of the operand types. When an **unsigned char** or **unsigned short** is “widened,” the result type is **int** if an **int** is large enough to represent all the values of the smaller type. Otherwise the result type would be an **unsigned int**. The *value preserving*, rule produces the “least surprise” arithmetic result for most expressions.

Compilation behavior

Only in the transition (-Xt) and K&R modes (-Xk) mode will the ANSI C compiler use the unsigned preserving promotions; in the other three modes, the value preserving promotion rules will be used. The compiler will warn you about each expression whose behavior might depend on the promotion rules used at compiler warning output levels at or above -W2.

This warning is not optional since this is a serious change in behavior. Fortunately, these situations do not often occur, and it is always possible to suppress the warning by making the intended behavior explicit, as is shown below.

First example: using a cast

In the following code, assume that an **unsigned char** is smaller than an **int**.

```
int f(void)
{
    int i = -2;
    unsigned char uc = 1;

    return (i + uc) < 17;
}
```

NOTE In the ANSI C compiler, the code above causes the compiler to issue the following warning:

```
line 6: warning: semantics of "<" change in ANSI C;
        use explicit cast
```

The result of the addition has type **int** (value preserving) or **unsigned int** (unsigned preserving), but the bit pattern does not change between these two. A two's-complement machine has:

```
i:  111...110  (-2)
+ uc: 000...001  ( 1)
=====
      111...111  (-1 or UINT_MAX)
```

This bit representation corresponds to -1 for **int** and **UINT_MAX** for **unsigned int**. Thus, if the result has type **int**, a signed comparison is used and the less-than test is true; if the result has type **unsigned int**, an unsigned comparison is used and the less-than test is false.

The addition of a cast serves to specify which of the two behaviors is desired:

value preserving:

```
(i + (int)uc) < 17
```

unsigned preserving:

```
(i + (unsigned int)uc) < 17
```

Because this expression can be viewed as ambiguous (since differing compilers chose different meanings for the same code), the addition of a cast is as much to help the reader as it is to eliminate the warning message.

Bit-fields

The same situation applies to the promotion of bit-field values. In ANSI C, if the number of bits in an **int** or **unsigned int** bit-field is less than the number of bits in an **int**, the promoted type is **int**; otherwise the promoted type is **unsigned int**. In most older C compilers, the promoted type is **unsigned int** for explicitly unsigned bit-fields, and **int** otherwise.

On machines where plain bit-fields represent unsigned values, full-sized *bit-fields* (for example, 8 bit chars, 16 bit shorts, 32 bit ints, longs, and enums) will be changed for the purpose of code generation to the corresponding unsigned type. Any full-sized bitfield, on any machine, will be changed to a simple type.

Similar use of casts can eliminate situations that are ambiguous.

Second example: result is the same

In the following code, assume that both **unsigned short** and **unsigned char** are narrower than **int**.

```
int f(void)
{
    unsigned short us;
    unsigned char uc;

    return uc < us;
}
```

In this example, both automatics are either promoted to **int** or to **unsigned int**, so the comparison is sometimes unsigned and sometimes signed. However, the ANSI C compiler will not warn you because the result is the same for the two choices.

Integral constants

As with expressions, the rules for the types of certain integral constants have changed. Previously, an unsuffixed decimal constant had type **int** only if its value fit in an **int** and an unsuffixed octal or hexadecimal constant had type **int** only if its value fit in an **unsigned int**. Otherwise, an integral constant had type **long**. (At times the value did not fit in the resulting type!) In ANSI C, the constant type is the first type encountered in the list below that corresponds to the value:

unsuffixed decimal:

int, long, unsigned long

unsuffixed octal or hexadecimal:

int, unsigned int, long, unsigned long

U suffixed: unsigned int, unsigned long

L suffixed: long, unsigned long

UL suffixed: unsigned long

The ANSI C compiler will warn you about any expression whose behavior might change according to the typing rules of the constants involved. The old integral constant typing rules are used only in the transition mode; the ANSI C and conforming modes use the new rules.

Third example: integral constants

In the following code, assume **ints** are 16 bits.

```
int f(void)
{
    int i = 0;

    return i > 0xffff;
}
```

Because the hexadecimal constant's type is either **int** (with a value of -1 on a two's-complement machine) or an **unsigned int** (with a value of 65535), the comparison will be true on a non-ANSI C compiler (**-Xt** and **-Xk** modes), and false on an ANSI C compiler (**-Xa**, **-Xc**, and **-Xm** modes).

An appropriate cast clarifies the code and silences the ANSI C compiler:

Non-ANSI C behavior:

```
i > (int)0xffff
```

ANSI C behavior:

```
i > (unsigned int)0xffff
or
i > 0xffffU
```

(The **U** suffix character is a new feature of ANSI C and will probably produce an error message with older compilers.)

Tokenization and preprocessing

Probably the least specified part of previous versions of C concerned the operations that transformed each source file from a bunch of characters into a sequence of tokens, ready to parse. These operations included recognition of white space (including comments), bundling consecutive characters into tokens, handling preprocessing directive lines, and macro replacement. However, their respective ordering was never guaranteed.

ANSI C translation phases

The order of these translation phases is specified by ANSI C:

1. Every “trigraph” sequence in the source file is replaced. ANSI C has exactly nine trigraph sequences that were invented solely as a concession to deficient character sets (as far as C is concerned) and are three-character sequences that name a character not in the ISO 646-1983 character set:

??=	#	??'	^
??-	~	??!	
??(	[	??/	\
??)	]		
??<	{		
??>	}		

These sequences must be understood by ANSI C compilers, but they should not be used except (possibly) to obscure code. The ANSI C compiler warns you whenever it replaces a trigraph while in transition or K&R mode, (-Xt or -Xk), even in comments. For example, consider the following:

```
/* comment *??/
/* still comment? */
```

The “??/” becomes a backslash. This character and the following newline are removed. The resulting characters are:

```
/* comment *//* still comment? */
```

The first “/” from the second line is the end of the comment. The next token is the “*”.

2. Every backslash/new-line character pair is deleted.
3. The source file is converted into preprocessing tokens and sequences of white space. Each comment is effectively replaced by a space character.
4. Every preprocessing directive is handled and all macro invocations are replaced. Each **#included** source file is run through the earlier phases before its contents replace the directive line.

5. Every escape sequence (in character constants and string literals) is interpreted.
6. Adjacent string literals are concatenated.
7. Every preprocessing token is converted into a (regular) token; the compiler proper parses these and generates code.
8. All external object and function references are resolved, resulting in the final program.

Old C translation phases

Older (pre-ANSI C compilers, such as previous versions of `rcc`, did not follow such a simple sequence of phases, nor were there any guarantees for when these steps were applied. A separate preprocessor recognized tokens and white space at essentially the same time as it replaced macros and handled directive lines. The output was then completely retokenized by the compiler proper, which then parsed the language and generated code.

Because the tokenization process within the preprocessor was a moment-by-moment thing and macro replacement was done as a character-based operation (and not token-based), the tokens and white space could have a great deal of variation during preprocessing.

There are several differences that arise from these two approaches. The rest of this section will discuss how code behavior may change due to line splicing, macro replacement, “stringizing,” and token “pasting,” which occur during macro replacement.

Logical source lines

In pre-ANSI C compilers, backslash/new-line pairs were allowed only as a means to continue a directive, a string literal, or a character constant to the next line. ANSI C extended the notion so that a backslash/new-line pair can continue anything to the next line. (The result is a “logical source line.”) Therefore, any code that relied on the separate recognition of tokens on either side of a backslash/new-line pair will not behave as expected.

Macro replacement

The macro replacement process has never been described in any significant detail prior to ANSI C. This vagueness spawned a great many divergent implementations and any code that relied on anything fancier than manifest constant replacement and simple `?:`-like macros was probably not truly portable. This tutorial cannot begin to uncover all the subtle and not so subtle differences between the old C macro replacement implementation and the ANSI C version. Fortunately, nearly all uses of macro replacement with the exception of token pasting and stringizing will produce exactly the same series of tokens as before. Furthermore, the ANSI C macro replacement algorithm can do things not possible in the old C version. For example:

```
#define name    (*name)
```

causes any use of *name* to be replaced with an indirect reference through *name*. (The old C preprocessor would produce a large amount of parentheses and stars and eventually complain about macro recursion.)

The major change in the macro replacement approach taken by ANSI C is to require macro arguments (other than those that are operands of the macro substitution operators `#` and `##`) to be expanded recursively prior to their substitution in the replacement token list. However, this change seldom produces an actual difference in the resulting tokens.

Stringizing

NOTE In ANSI C, the examples below marked with a ‡ produce a warning about use of old features. Only in the transition and K&R modes (`-Xt` and `-Xk`) will the result be the same as in previous versions of C.

In pre-ANSI C compilers, the following produced the string literal "x y!":

```
#define str(a) "a!" ‡
str(x y)
```

Thus the preprocessor searched inside string literals (and character constants) for characters that looked like macro parameters. ANSI C recognized the importance of this feature, but could not condone operations on parts of tokens. (In ANSI C, all invocations of the above macro produce the string literal "a!".) To achieve the old effect in ANSI C, we make use of the `#` macro substitution operator and the concatenation of string literals:

```
#define str(a) #a "!"
str(x y)
```

The above produces the two string literals "x y" and "!" which, after concatenation, produces the identical "x y!".

Unfortunately, there is no direct replacement for the analogous operation for character constants. The major use of this feature was similar to the following:

```
#define CNTL(ch) (037 & 'ch')      ‡
CNTL(L)
```

which produced

```
(037 & 'L')
```

which evaluates to the ASCII control-L character. The best solution is to change all uses of this macro (at least this can be done automatically) to:

```
#define CNTL(ch) (037 & (ch))
CNTL('L')
```

which is arguably more readable and more useful, as it can also be applied to expressions.

Token pasting

In pre-ANSI C compilers, there were at least two ways to combine two tokens. Both invocations in the following produced a single identifier `x1` out of the two tokens `x` and `1`.

```
#define self(a) a
#define glue(a,b) a/**/b      ‡
self(x)1
glue(x,1)
```

ANSI C could not sanction either approach to what they believed to be an important capability. In ANSI C, both the above invocations would produce the two separate tokens `x` and `1`. The second of the above two methods can be rewritten for ANSI C by using the `##` macro substitution operator:

```
#define glue(a,b) a ## b
glue(x, 1)
```

Since `##` is an actual operator, the invocation can be much freer with respect to white space in both the definition and invocation.

There is no direct approach to effect the first of the two old-style pasting schemes, but since it put the burden of the pasting at the invocation, it was used less frequently than the other form.

NOTE `#` and `##` should be used as macro substitution operators only when `__STDC__` is defined.

Using **const** and **volatile**

The keyword **const** is one of the C++ features that is found in ANSI C. When an analogous keyword, **volatile**, was invented by the ANSI C committee, in effect, the *type qualifier* category was created. This still remains one of the more nebulous parts of ANSI C.

Types for lvalues

const and **volatile** are part of an identifier's type, not its storage class. However, they are peculiar in that they are often removed from the top-most part of the type. This occurs when an object's value is fetched in the evaluation of an expression — exactly at the point when an lvalue becomes an rvalue. (These terms arise from the prototypical assignment "**L**=**R**"; in which the left side must still refer directly to an object (an lvalue) and the right side need only be a value (an rvalue).) Therefore, only expressions that are lvalues can be qualified by **const** or **volatile** or both.

Type qualifiers in derived types

The type qualifiers are unique in that they may modify type names and derived types. Derived types are those parts of C's declarations that can be applied over and over to build more and more complex types: pointers, arrays, functions, structures, and unions. Except for functions, one or both type qualifiers can be used to change the behavior of a derived type.

For example:

```
const int five = 5;
```

declares and initializes an object with type **const int** whose value will not be changed by a correct program. (The order of the keywords is not significant to C. For example, the declarations:

```
int const five = 5;
```

and

```
const five = 5;
```

are identical to the above declaration in its effect.)

The declaration:

```
const int *pci = &five ;
```

declares an object with type pointer to **const int**, which initially points to the previously declared object. Note that the pointer itself does not have a qualified type — it points to a qualified type, and as such, the pointer can be changed to point to essentially any **int** during the program's execution, but **pci** cannot be used to modify the object to which it points unless a cast is used, as in the following:

```
*(int *)pci = 17;
```

(If **pci** actually points to a **const object**, the behavior of this code is undefined.)

The declaration:

```
extern int *const cpi;
```

says that somewhere in the program there exists a definition of a global object with type **const** pointer to **int**. In this case, **cpi**'s value will not be changed by a correct program, but it can be used to modify the object to which it points. Notice that **const** comes after the ***** in the above declaration. The following pair of declarations produces the same effect:

```
typedef int *INT_PTR;
extern const INT_PTR cpi;
```

These can be combined as in the following declaration in which an object is declared to have type **const** pointer to **const int**:

```
const int *const cpci;
```

Using const to read character values

In hindsight, **readonly** would have been a better choice for a keyword than **const**. If one reads **const** in this manner, declarations such as:

```
char *strcpy(char *, const char *);
```

are easily understood to mean that the second parameter will only be used to read character values, while the first parameter will undoubtedly overwrite the characters to which it points. Furthermore, despite the fact that in the above example the type of **cpi** is a pointer to a **const int**, you can still change the value of the object to which it points through some other means (unless it actually points to an object declared with **const int** type).

Examples of **const** usage

The two main uses for **const** are to declare (large) compile-time initialized tables of information as unchanging, and to specify that pointer parameters will not modify the objects to which they point.

The first use potentially allows portions of the data for a program to be shared by other concurrent invocations of the same program (just as the code for the program can be), and may cause attempts to modify this presumably invariant data to be detected immediately by means of some sort of memory protection fault, as the data resides in a read-only portion of memory.

The second use will most likely help locate potential errors (before generating a memory fault during that critical demo). For example, functions that temporarily place a null character into the middle of a string will be detected at compile time if passed a pointer to a string that cannot be so modified.

When to use **volatile**

The previous examples have all used **const** because it's conceptually simpler. However, when using exact semantics is necessary, **volatile** should be used. To the programmer, **volatile** has multiple meanings. To a compiler writer it means to take no code generation shortcuts when accessing such an object. Moreover in ANSI C, it is a programmer's responsibility to declare every object that has the appropriate special properties with a **volatile**-qualified type.

Examples of **volatile** usage

The usual four examples of **volatile** objects are:

1. an object that is a memory-mapped I/O port
2. an object that is shared between multiple concurrent processes
3. an object that is modified by an asynchronous signal handler
4. an automatic storage duration object declared in a function that calls **setjmp** and whose value is changed between the call to **setjmp** and a corresponding call to **longjmp**

The first three examples are all instances of an object with a particular behavior whose value can be modified at any point during the execution of the program. Thus, the seemingly infinite loop:

```
flag = 1;
while (flag)
    ;
```

is completely reasonable as long as **flag** has a **volatile**-qualified type. (Presumably, some asynchronous event will set **flag** to zero in the future.) Otherwise, the compilation system is free to change the above loop (because

the value of **flag** is unchanged within the body of the loop) into a truly infinite loop that completely ignores the value of **flag**.

The fourth example, involving variables local to functions that call **setjmp**, is more involved. If you read the fine print about the behavior of **setjmp** and **longjmp**, you will find there are no guarantees about the values for objects matching the fourth case. It turns out to be necessary for **longjmp** to examine every stack frame between the function calling **setjmp** and the function calling **longjmp** for saved register values in order to get the most desirable behavior. The possibility of asynchronously created stack frames makes this expensive job even harder. Therefore, most implementations just documented the undesirable side effect and used an inexpensive implementation.

When an automatic object is declared with a **volatile**-qualified type, the compilation system knows that it has to produce code that exactly matches what the programmer wrote. Therefore, the most recent value for such an automatic object will always be in memory (not just in a register) and as such will be guaranteed to be up-to-date when **longjmp** is called.

Multibyte characters and wide characters

At first, the “internationalization” of ANSI C affected only library functions. However the final stage of internationalization — multibyte characters and wide characters — also affected the language proper.

“Asianization” means multibyte characters

The basic difficulty in an Asian environment is the huge number of ideograms needed for I/O. To work within the constraints of usual computer architectures, these ideograms are encoded as sequences of bytes. The associated operating systems, application programs, and terminals understand these byte sequences as individual ideograms. Moreover, all of these encodings allow intermixing of regular single-byte characters with the ideogram byte sequences. Just how difficult it is to recognize distinct ideograms depends on the encoding scheme used.

The term “multibyte character” is defined by ANSI C to denote a byte sequence that encodes an ideogram, no matter what encoding scheme is employed. All multibyte characters are members of the so-called “extended character set.” (A regular single-byte character is just a special case of a multibyte character.) Essentially the only requirement placed on the encoding is that no multibyte character can use a null character as part of its encoding.

ANSI C specifies that program comments, string literals, character constants, and header names are all sequences of multibyte characters.

Encoding variations

The encoding schemes come in two variations:

1. Where each multibyte character is self-identifying, therefore, any multibyte character can simply be inserted between any pair of multibyte characters. (The encoding used by the ANSI C compiler is one of these types; each byte of a non-single-byte character has the high-order bit set.)
2. Where the presence of special *shift bytes* changes the interpretation of subsequent bytes. An example is the method used by most fancy character terminals to get in and out of line drawing mode. For programs written in multibyte characters with a shift-state-dependent encoding, ANSI C has the additional requirement that each comment, string literal, character constant, and header name must both begin and end in the unshifted state.

Wide characters

Some of the inconvenience of handling multibyte characters would be eliminated if all characters were of a uniform number of bytes or bits. Since there can be thousands or tens of thousands of ideograms in such a character set, a 16-bit or 32-bit sized integral value should be used to hold all members.

NOTE The full Chinese alphabet includes more than 65,000 ideograms.

ANSI C includes the **typedef** name **wchar_t** as the implementation-defined integral type large enough to hold all members of the extended character set.

For each wide character there is a corresponding multibyte character and vice versa; the wide character that corresponds to a regular single-byte character is required to have the same value as its single-byte value, including the null character. However, there is no guarantee that the value of the macro **EOF** can be stored in a **wchar_t**. (Just as **EOF** might not be representable as a **char**.)

Conversion functions

ANSI C provides five library functions that manage multibyte characters and wide characters:

mblen()	length of next multibyte character
mbtowc()	convert multibyte character to wide character
wctomb()	convert wide character to multibyte character
mbstowcs()	convert multibyte character string to wide character string
wcstombs()	convert wide character string to multibyte character string

The behavior of all of these functions depends on the current locale.

NOTE See the `setlocale()` function in “Internationalization” (page 45).

It is expected that vendors providing compilation systems targeted to this market will supply many more string-like functions to simplify the handling of wide character strings. However, for most application programs, there is no need to convert any multibyte characters to or from wide characters. Programs such as **diff**, for example, will read in and write out multibyte characters, needing only to check for an exact byte-for-byte match. More complicated programs (such as **grep**) that use regular expression pattern matching, may need to understand multibyte characters, but only the common set of functions that manages the regular expression needs this knowledge. The program **grep** itself requires no other special multibyte character handling.

C language features

To give even more flexibility to the programmer in an Asian environment, ANSI C provides wide character constants and wide string literals. These have the same form as their non-wide versions except that they are immediately prefixed by the letter **L**:

<code>'x'</code>	regular character constant
<code>'¥'</code>	regular character constant
<code>L'x'</code>	wide character constant
<code>L'¥'</code>	wide character constant
<code>"abc¥xyz"</code>	regular string literal
<code>L"abc¥xyz"</code>	wide string literal

Notice that multibyte characters are valid in both the regular and wide versions. The sequence of bytes necessary to produce the ideogram ¥ is encoding-specific, but if it consists of more than one byte, the value of the character constant `'¥'` is implementation defined, just as the value of `'ab'` is implementation defined. A regular string literal contains exactly the bytes (except for escape sequences) specified between the quotes, including the bytes of each specified multibyte character.

When the compilation system encounters a wide character constant or wide string literal, each multibyte character is converted (as if by calling the `mbtowc()` function) into a wide character. Thus, the type of `L'¥'` is `wchar_t` the type of `L"abc¥xyz"` is array of `wchar_t` with length eight. (Just as with regular string literals, each wide string literal has an extra zero-valued element appended, but in these cases it is a `wchar_t` with value zero.)

Just as regular string literals can be used as a short-hand method for character array initialization, wide string literals can be used to initialize `wchar_t` arrays:

```
wchar_t *wp = L"a%z";  
wchar_t x[] = L"a%z";  
wchar_t y[] = {L'a', L'%', L'z', 0};  
wchar_t z[] = {'a', L'%', 'z', '\\0'};
```

In the above example, the three arrays `x`, `y`, and `z`, and the array pointed to by `wp`, have the same length and all are initialized with identical values.

Finally, adjacent wide string literals will be concatenated, just as with regular string literals. However, adjacent regular and wide string literals produce undefined behavior. A compiler is not even required to complain if it does not accept such concatenations.

Standard headers and reserved names

Very early in the standardization process, the ANSI Standards Committee chose to include library functions, macros, and header files as part of ANSI C. These macros, header files, and functions are necessary to write truly portable C programs, however, they also produce a large set of reserved names.

This section presents the various categories of reserved names and some rationale for their reservations. At the end is a set of rules to follow that can steer your programs clear of any reserved names.

Balancing process

In order to match existing implementations, the ANSI C committee had to choose names like `printf` and `NULL`; to have done otherwise would have disqualified virtually all existing C programs from conformance, and would obviously run counter to their charter to standardize existing practice. However, each such name reduced the set of names available for free use in C programs.

On the other hand, before standardization, implementors felt free to add both new keywords to their compilers and names to headers. This meant that no program could be guaranteed to compile from one release to another, let alone port from one vendor's implementation to another.

Thus, the Committee made a hard decision: to restrict all conforming implementations from including any extra names, except those with certain forms. It is this decision, more than any other, that will cause most C compilation systems to be almost conforming. Nevertheless, the Standard contains 32 keywords and almost 250 names in its headers, none of which necessarily follow any particular naming pattern.

Standard headers

The standard headers are as follows:

<i>assert.h</i>	<i>locale.h</i>	<i>stddef.h</i>
<i>ctype.h</i>	<i>math.h</i>	<i>stdio.h</i>
<i>errno.h</i>	<i>setjmp.h</i>	<i>stdlib.h</i>
<i>float.h</i>	<i>signal.h</i>	<i>string.h</i>
<i>limits.h</i>	<i>stdarg.h</i>	<i>time.h</i>

Most implementations will provide more headers, but a strictly conforming ANSI C program can only use these.

Other standards disagree slightly regarding the contents of some of these headers. For example, POSIX (IEEE 1003.1) specifies that **fdopen** is declared in *stdio.h*. To allow these two standards to coexist, POSIX requires the macro **_POSIX_SOURCE** to be **#defined** prior to the inclusion of any header to guarantee that these additional names exist. (In actuality, the POSIX committee believes that almost the opposite will occur: that **_POSIX_SOURCE** will be used to limit headers to only those names POSIX describes, and that, by default, headers will contain even more names than POSIX specifies.) X/Open, in its *Portability Guide*, has also used this macro scheme for its extensions. X/Open's macro is **_XOPEN_SOURCE**. The following section describes why this scheme is sufficient.

ANSI C requires the standard headers to be both self-sufficient and idempotent. No standard header needs any other header to be **#included** before or after it, and each standard header can be **#included** more than once without causing problems. The Standard also requires that its headers be **#included** only in safe contexts so that the names used in the headers are guaranteed to remain unchanged.

Names reserved for implementation use

The Standard places further restrictions on implementations regarding their libraries. While in the past, most programmers learned not to use names like **read** and **write** for their own functions on UNIX systems (usually after encountering interesting program behavior), ANSI C requires that only names reserved by the Standard be introduced by references within the implementation.

Thus, the Standard reserves a subset of all possible names for implementations to use as they so choose. This class of names consists of identifiers that begin with an underscore and continue with either another underscore or a capital letter. The class of names contains all names matching the following regular expression:

```
_[_A-Z][0-9_a-zA-Z]*
```

Strictly speaking, if your program uses such an identifier, its behavior is undefined. Thus, programs using `_POSIX_SOURCE` (or `_XOPEN_SOURCE`) have undefined behavior.

However, undefined behavior comes in different degrees. If, in a POSIX-conforming implementation you use `_POSIX_SOURCE`, you know that your program's "undefined behavior" consists of certain additional names in certain headers, and your program still conforms to an accepted standard. This deliberate loophole in the ANSI C standard allows implementations to conform to seemingly incompatible specifications. On the other hand, an implementation that does not conform to the POSIX standard is free to behave in any manner when encountering a name such as `_POSIX_SOURCE`.

The Standard also reserves all other names that begin with an underscore for use in header files as regular file scope identifiers and as tags for structures and unions, but not in local scopes. This means that the common existing practice of having functions named `_filbuf` and `_doprint` to implement hidden parts of the library is sanctioned.

Names reserved for expansion

In addition to all the names explicitly reserved, the Standard also reserves (for implementations and future standards) names matching certain patterns:

<i>errno.h</i>	<code>E[0-9A-Z].*</code>
<i>ctype.h</i>	<code>(to is)[a-z].*</code>
<i>locale.h</i>	<code>LC_[A-Z].*</code>
<i>math.h</i>	<i>current function names</i> <code>[fl]</code>
<i>signal.h</i>	<code>(SIG SIG_)[A-Z].*</code>
<i>stdlib.h</i>	<code>str[a-z].*</code>
<i>string.h</i>	<code>(str mem wcs)[a-z].*</code>

In the above lists, names that begin with a capital letter are macros and are reserved only when the associated header is included. The rest of the names designate functions and therefore cannot be used to name any global objects or functions.

Names safe to use

As you can tell by now, the rules regarding when certain names are reserved are complicated. There are, however, four fairly simple rules you can follow to keep from colliding with any ANSI C reserved names:

1. **#include** all system headers at the top of your source files (except possibly after a **#define** of `_POSIX_SOURCE` or `_XOPEN_SOURCE`, or both).
2. Do not define or declare any names that begin with an underscore.
3. Use an underscore or a capital letter somewhere within the first few characters of all file scope tags and regular names. (But beware of the `va_` prefix found in `stdarg.h` or `varargs.h`.)
4. Use a digit or a non-capital letter somewhere within the first few characters of all macro names. (But note that almost all names beginning with an "E" are reserved if `errno.h` is **#included**.)

As noted earlier, most implementations will continue to add names to the standard headers by default. Therefore, these rules are just a guideline to follow.

Internationalization

A previous section introduced the "internationalization" of the standard libraries. (See "Multibyte characters and wide characters" (page 39)). This section discusses the affected library functions and gives some hints on how programs should be written to take advantage of these features.

Locales

At any time, a C program has a current "locale" — a collection of information that describes the conventions appropriate to some nationality, culture, and language. Locales have names that are strings and the only two standardized locale names are "C" and "". In source-level APIs other than the `ods30` environment (`-a ods30`), each program begins in the "C" locale which unsurprisingly causes all library functions to behave just like they have historically. The "" locale is the implementation's best guess at the correct set of conventions appropriate to the program's invocation. (Of course "C" and "" can cause identical behavior.) Other locales may be provided by implementations.

For the purposes of practicality and expediency, locales are partitioned into a set of categories. A program can change the complete locale (all categories) or one or more categories leaving the other categories unchanged. Generally each category affects a set of functions disjoint from the functions affected by other categories, so temporarily changing one category for a little while can make sense.

The **setlocale()** function

The **setlocale()** function is the interface to the program's locale. In general, any program that requires the invocation country's conventions should place a call such as:

```
#include <locale.h>
/*...*/
setlocale(LC_ALL, "");
```

early in the program's execution path. This causes the program's current locale to change to the appropriate local version (if possible), since **LC_ALL** is the macro that specifies the entire locale instead of one category. The following are the standard categories:

LC_COLLATE	sorting information
LC_CTYPE	character classification information
LC_MONETARY	currency printing information
LC_NUMERIC	numeric printing information
LC_TIME	date and time printing information
LC_MESSAGES	message language information

Any of these macros can be passed as the first argument to **setlocale()** to specify just that category.

The **setlocale()** function returns the name of the current locale for a given category (or **LC_ALL**) and serves in an inquiry-only capacity when its second argument is a null pointer. Thus, code along the lines of the following can be used to change the locale or a portion thereof for a limited duration:

```
#include <locale.h>
/*...*/
char *oloc;
/*...*/
oloc = setlocale(LC_cat, NULL);
if (setlocale(LC_cat, "new") != 0)
{
    /* use temporarily changed locale */
    (void)setlocale(LC_cat, oloc);
}
```

Most programs will never need this capability.

Changed functions

Wherever possible and appropriate, existing library functions were extended to include locale-dependent behavior. These functions came in two groups: those declared by the *ctype.h* header (character classification and conversion), and those functions that convert to and from printable and internal forms of numeric values, for example, **printf()** and **strtod()**.

All *ctype.h* predicate functions except **isdigit()** and **isxdigit()** are allowed to return nonzero (true) for additional characters when the **LC_CTYPE** category of the current locale is other than "C". In a Spanish locale, **isalpha('ñ')** should be true. Similarly the character conversion functions **tolower()** and **toupper()** should appropriately handle any extra alphabetic characters identified by the **isalpha()** function. (As an implementation note, the *ctype.h* functions are almost always macros that are implemented using table lookups indexed by the character argument. Their behavior is changed by resetting the table(s) to the new locale's values, and therefore there is no performance impact.)

Those functions that write or interpret printable floating values may change to use a decimal-point character other than period (.) when the **LC_NUMERIC** category of the current locale is other than "C". There is no provision for converting any numeric values to printable form with thousands separator-type characters, but when converting from a printable form to an internal form, implementations are allowed to accept such additional forms, again in other than the "C" locale. Those functions that make use of the decimal-point character are the **printf()** and **scanf()** families, **atof()**, and **strtod()**. Those functions that are allowed implementation-defined extensions are **atof()**, **atoi()**, **atol()**, **strtod()**, **strtol()**, **strtoul()**, and the **scanf()** family. (ANSI C currently defines no such extensions.)

New functions

Certain locale-dependent capabilities were added as new standard functions. Besides **setlocale()** which allows control over the locale itself, the Standard includes the following new functions and the multibyte functions previously discussed (**mblen()**, **mbtowc()**, **mbstowcs()**, **wctomb()**, and **wcstombs()**):

localeconv()	numeric/monetary conventions
strcoll()	collation order of two strings
strxfrm()	translate string for collation
strftime()	formatted date/time conversion

The **localeconv()** function returns a pointer to a structure containing information useful for formatting numeric and monetary information appropriate to the current locale's **LC_NUMERIC** and **LC_MONETARY** categories. (This is the only function whose behavior depends on more than one category.) For numeric values the structure describes the decimal-point character, the thousands separator, and where the separator(s) should be located. There are fifteen other structure members that describe how to format a monetary value.

The **strcoll()** function is analogous to the **strcmp()** function except that the two strings are compared according to the **LC_COLLATE** category of the current locale. As this comparison is not necessarily as inexpensive as **strcmp()**, the **strxfrm()** function can be used to transform a string into another, such that any two such after-translation strings can be passed to **strcmp()** and get an ordering analogous to what **strcoll()** would have returned if passed the two pre-translation strings.

The **strftime()** function provides a **sprintf()**-like formatting of the values in a **struct tm**, along with some date and time representations that depend on the **LC_TIME** category of the current locale. This function is based on the function **asctime(S)**.

Grouping and evaluation in expressions

One of the choices made by Dennis Ritchie in the design of C was to give compilers license to rearrange expressions involving adjacent operators that are mathematically commutative and associative, even in the presence of parentheses. This was explicitly noted in the “Reference Manual Appendix” of *The C Programming Language*, Kernighan and Ritchie, (First Edition). ANSI C does not grant compilers this same freedom.

This section discusses the differences between these two definitions of C and clarifies the distinctions between an expression's side effects, grouping, and evaluation by considering the expression statement from the following code fragment.

```
int i, *p, f(void), g(void);
/*...*/
i = ++p + f() + g();
```

Definitions

The side effects of an expression are its modifications to memory and its accesses to **volatile**-qualified objects. The side effects in the above expression are the updating of **i** and **p** and any side effects contained within the functions **f()** and **g()**.

An expression's "grouping" is the way values are combined with other values and operators. The above expression's grouping is, primarily, the order in which the additions are performed.

An expression's "evaluation" includes everything necessary to produce its resulting value. To evaluate an expression, all specified side effects must occur (anywhere between the previous and next sequence point) and the specified operations are performed with a particular grouping. For the above expression, the updating of *i* and *p* must occur after the previous statement and by the ";" of this expression statement; the calls to the functions can occur in either order, any time after the previous statement, but before their return values are used. In particular, note that the operators that cause memory to be updated have no requirement to assign the new value before the value of the operation is used.

The Kernighan and Ritchie C rearrangement license

The Kernighan and Ritchie C rearrangement license applies to the above expression because addition is mathematically commutative and associative. To distinguish between regular parentheses and the actual grouping of an expression, the left and right curly braces will designate grouping. The three possible groupings for the expression are:

```
i = { {++p + f()} + g() };
i = { ++p + {f() + g()} };
i = { {++p + g()} + f() };
```

all of which are valid given Kernighan and Ritchie C rules. Moreover, all of these groupings are valid even if the expression were written instead, for example, in either of these ways:

```
i = ++p + (f() + g());
i = (g() + ++p) + f();
```

If this expression is evaluated on an architecture for which either overflows cause an exception or addition and subtraction are not inverses across an overflow, these three groupings will behave differently if one of the additions overflows.

For such expressions on these architectures, the only recourse available in Kernighan and Ritchie C was to split the expression to force a particular grouping. The following are possible rewrites that respectively enforce the above three groupings.

```
i = ++p; i += f(); i += g();
i = f(); i += g(); i += ++p;
i = ++p; i += g(); i += f();
```

The ANSI C rules

ANSI C does not allow operations to be rearranged that are mathematically commutative and associative, but that are not actually so on the target architecture. Thus the precedence and associativity of the ANSI C grammar completely describes the grouping for all expressions; all expressions must be grouped as they are parsed. The expression under consideration is grouped in this manner:

```
i = { (*++p + f()) + g() };
```

(This still does not mean that `f()` must be called before `g()`, or that `p` must be incremented before `g()` is called.)

In ANSI C, expressions need not be split to guard against unintended overflows.

Parentheses grouping and evaluation

ANSI C is often erroneously described as honoring parentheses or evaluating according to parentheses due to an incomplete understanding or an inaccurate presentation.

Since ANSI C expressions simply have the grouping specified by their parsing, parentheses still only serve as a way of controlling how an expression is parsed; the natural precedence and associativity of expressions carry exactly the same weight as parentheses.

The above expression could have been written as:

```
i = (((*++p)) + f()) + g();
```

with no different effect on its grouping or evaluation.

The “as if” rule

There were good reasons for the Kernighan and Ritchie C rearrangement rules:

- The rearrangements provide many more opportunities for optimizations such as compile-time constant folding.
- The rearrangements do not change the result of integral-typed expressions on most machines.
- Some of the operations are both mathematically and computationally commutative and associative on all machines.

The ANSI C committee eventually became convinced that the rearrangement rules were intended to be an instance of the “as if” rule when applied to the described target architectures. ANSI C’s “as if” rule is a general license that permits an implementation to deviate arbitrarily from the abstract machine description as long as the deviations do not change the behavior of a valid C program.

Thus all the binary bitwise operators (other than shifting) are allowed to be rearranged on any machine because there is simply no way to notice such regroupings. On typical two’s complement machines in which overflow silently wraps around, integer expressions involving multiplication or addition can be rearranged for the same reason.

Therefore, this change in C does not have a significant impact on most C programmers.

Incomplete types

The ANSI C standard introduced the term “incomplete type” to formalize a fundamental, yet misunderstood, portion of C, implicit from its beginnings. This article describes incomplete types, where they are permitted, and why they are useful.

Types

ANSI separates C’s types into three distinct sets: function, object, and incomplete. Function types are obvious; object types cover everything else, except when the size of the object is not known. The Standard uses the term “object type” to specify that the designated object must have a known size, but it is important to know that incomplete types other than **void** also refer to an object.

There are only three variations of incomplete types: **void**, arrays of unspecified length, and structures and unions with unspecified content. The type **void** differs from the other two in that it is an incomplete type that cannot be completed, and it serves as a special function return and parameter type.

Completing incomplete types

An array type is completed by specifying the array size in a following declaration in the same scope that denotes the same object. (Also, when an array without a size is declared and initialized in the same declaration, the array has an incomplete type only between the end of its declarator and the end of its initializer.)

An incomplete structure or union type is completed by specifying the content in a following declaration in the same scope for the same tag.

Declarations

Certain declarations can use incomplete types, but others require (complete) object types. Those declarations that require object types are array elements, members of structures or unions, and objects local to a function. All other declarations permit incomplete types. In particular, the following are permitted:

- pointers to incomplete types
- functions returning incomplete types
- incomplete function parameter types
- **typedef** names for incomplete types

The function return and parameter types are special. Except for **void**, an incomplete type used in such a manner must be completed by the time the function is defined or called. (A return type of **void** specifies a function that returns no value and a single parameter type of **void** specifies a function that accepts no arguments.)

Note that since array and function parameter types are rewritten to be pointer types, a seemingly incomplete array parameter type is not actually incomplete. The typical declaration of **main**'s **argv** (namely, **char *argv[]**) as an unspecified length array of character pointers, is rewritten to be a pointer to character pointers.

Expressions

Most expression operators require (complete) object types. The only three exceptions are the unary **&** operator, the first operand of the comma operator, and the second and third operands of the **"?:"** operator. Most operators that accept pointer operands also permit pointers to incomplete types, unless pointer arithmetic is required. The list includes the unary **"*"** operator, even though some may find this surprising. For example, given:

```
void *p
```

&*p is a valid subexpression that makes use of this.

Justification

C would have been simpler without incomplete types. However, they are necessary for **void**, and there is one feature provided by incomplete types that C has no other way to handle, and that has to do with forward references to structures and unions. If one has two structures that need pointers to each other, the only way to do so (without resorting to potentially invalid casts) is with incomplete types:

```
struct a { struct b *bp; };
struct b { struct a *ap; };
```

All strongly typed programming languages that have some form of pointer and heterogeneous data types provide some method of handling this case.

Examples

Defining **typedef** names for incomplete structure and union types is frequently quite useful. If one has a complicated bunch of data structures that contain many pointers to each other, having a list of **typedefs** to the structures up front (possibly in a central header) can simplify the declarations.

```
typedef struct item_tag Item;
typedef union  note_tag Note;
typedef struct list_tag List;
. . .
struct item_tag { . . . };
. . .
struct list_tag {
    List *next; . . .
};
```

Moreover, for those structures and unions whose contents should not be available to the rest of the program, a header can declare the tag without the content. Other parts of the program can use pointers to the incomplete structure or union without any problems (unless they attempt to use any of its members).

A frequently used incomplete type is an external array of unspecified length. It generally is not necessary to know the extent of an array to make use of its contents:

```
extern char *tzname[];
. . .
(void)fputs("Alternate time zone: ", stdout);
(void)puts(tzname[1]);
```

Compatible and composite types

With Kernighan and Ritchie C, and even more so with ANSI C, it is possible for two declarations that refer to the same entity to be other than identical. The term “compatible type” is used in ANSI C to denote those types that are “close enough.” This section describes compatible types as well as “composite types” — the result of combining two compatible types.

Multiple declarations

If a C program were only allowed to declare each object or function once, there would be no need for compatible types. But linkage (which allows two or more declarations to refer to the same entity), function prototypes, and separate compilation all need such a capability. Not too surprisingly, separate translation units (source files) have different rules for type compatibility than within a single translation unit.

Separate compilation compatibility

Since each compilation probably looks at different source files, most of the rules for compatible types across separate compiles are structural in nature:

- Matching scalar (integral, floating, and pointer) types must be compatible, as if they were in the same source file.
- Matching structures, unions, and enums must have the same number of members and each matching member must have a compatible type (in the separate compilation sense), including bit-field widths.
- Matching structures must have the members in the same order. (The order of union and enum members does not matter.)
- Matching enum members must have the same value.

An additional requirement is that the names of members (including the lack of names for unnamed members) match for structures, unions, and enums, but not necessarily their respective tags.

Single compilation compatibility

When two declarations in the same scope describe the same object or function, the two declarations must specify compatible types. These two types are then combined into a single composite type that is compatible with the first two. More about composite types later.

The compatible types are defined recursively. At the bottom are type specifier keywords. (These are the rules that say that **unsigned short** is the same as **unsigned short int**, and that a type without type specifiers is the same as one

with **int**.) All other types are compatible only if the types from which they are derived are compatible. For example, two qualified types are compatible if the qualifiers (**const** and **volatile**) are identical and the unqualified base types are compatible.

Compatible pointer types

For two pointer types to be compatible, the types they point to must be compatible and the two pointers must be identically qualified. Recall that the qualifiers for a pointer are specified after the *****, so that these two declarations:

```
int *const cpi;
int *volatile vpi;
```

declare two differently qualified pointers to the same type, **int**.

Compatible array types

For two array types to be compatible, their element types must be compatible, and, if both array types have a specified size, they must match. This last part means that an incomplete array type (see “Incomplete types” (page 51)) is compatible both with another incomplete array type and an array type with a specified size.

Compatible function types

For two function types to be compatible, their return types must be compatible. If either or both function types have prototypes, the rules get more complicated.

For two function types with prototypes to be compatible, they also must have the same number of parameters (including use of the ellipsis (**...**) notation) and the corresponding parameters must be parameter-compatible.

For an old style function definition to be compatible with a function type with a prototype, the prototype parameters must not end with an ellipsis (**...**) and each of the prototype parameters must be parameter-compatible with the corresponding old style parameter, after application of the default argument promotions.

For an old style function declaration (not a definition) to be compatible with a function type with a prototype, the prototype parameters must not end with an ellipsis (**...**) and all of the prototype parameters must have types that would be unaffected by the default argument promotions.

For two types to be parameter-compatible, the types must be compatible after the (top level) qualifiers, if any, have been removed, and after a function or array type has been converted to the appropriate pointer type.

Special cases

There are a few surprises in this area. For example, **signed int** behaves the same as **int** except possibly for bit-fields, in which a plain **int** may denote an unsigned-behaving quantity.

Another interesting note is that each enumeration type must be compatible with some integral type. For portable programs this means that enumeration types effectively are separate types, and, for the most part, the ANSI C standard views them in that manner.

Composite type

The construction of a composite type from two compatible types is also recursively defined. The ways compatible types can differ from each other are due either to incomplete arrays or to old style function types. As such, the simplest description of the composite type is that it is the type compatible with both of the original types, including every available array size and every available parameter list from the original types.

Chapter 3

Constructing and using message catalogs

In an effort to better internationalize application software, message catalogs are used in place of hard-coded messages within the actual code. All input, output, and error messages are precompiled and stored in message catalogs that are outside of the code. Using message catalogs, the user need not change an application's hard-coded messages and be forced to recompile when a language change is required. Instead, the program will present all messages in a specified language according to the values of certain environment variables. A language change would involve a simple change to an environment variable at runtime. Needless to say, message catalogs allow greater flexibility to users as well as provide an alternative to the often difficult task of modifying hard-coded applications written for another language.

A message catalog is created by a message source file using the **gencat**(CP) command. They can be accessed and used within software using the **catopen**(S), **catgets**(S), and the **catclose**(S) routines.

For more background about message catalogs or internationalizing code in general, see the Open Software Foundation's *Internationalization Made Easy* White Paper.

The following example code gives an example of how to construct a simple message catalog and then how to incorporate it into a program. To understand the example code, you should understand the role of environment variables; particularly the variables **LC_MESSAGES**, **LC_ALL**, **LANG**, and **NLSPATH**. The precise meanings of these variables are defined in **setlocale**(S).

Creating a message catalog

Before you can use message catalogs within programs, they need to be constructed first. The following two code fragments are the message source files for the same message in the English and French languages respectively.

Example 3-1 English language message catalog (english/hello.text)

```
1 $set 1
2 1 Hello everyone. It's
3 2 %a %b %d %T %y
```

- | | |
|--------|--|
| line 1 | Each message is identified by a set number and then a message number. This line defines the set number to be 1. |
| line 2 | This line defines the message number to be 1 also. Therefore the message is identified as message 1 in set 1. Following the message id is the actual body of the message. Quotation marks are not necessary. |
| line 3 | Similarly, this line defines the message number to be 2, identifying this as message 2 in set 1. |

Example 3-2 French language message catalog (french/hello.text)

```
1 $set 1
2 1 Bonjour tout le monde. C'est
3 2 %a %d %b %Hh%M %Y
```

- | | |
|-----------|---|
| lines 1-3 | These lines serve the same purpose as in the first example except the body of the message is in French. Thus, if the French language was used, this message would be printed instead. |
|-----------|---|

Code example using a message catalog

The following code uses the message just created in “Creating a message catalog” (page 58). Rather than hard-coding output directly, it reads the proper message catalog depending on the `LANG` environment variable.

Example 3-3 Internationalized “Hello, world” program (hello.c)

```

1 #include <stdio.h>
2 #include <time.h>
3 #include <nl_types.h>
4 #include <langinfo.h>
5 #include <locale.h>

6 #define TEMPBUF_SIZE    100

7 main()
8 {
9     nl_catd catalog;
10    time_t clock;
11    char    tempbuffer[TEMPBUF_SIZE];

12    setlocale (LC_ALL, "");

13    if ((catalog = catopen ("hello.cat", 0)) == -1) {
14        fprintf(stderr, "Can't open message catalog\n");
15        /* Continue anyway, this is not fatal */
16    }

17    (void) time(&clock);

18    (void) strftime(tempbuffer, TEMPBUF_SIZE,
19                    catgets(catalog, 1, 2, "%a %b %T %y"),
20                    localtime(&clock));
21    printf("%s %s\n",
22          catgets(catalog, 1, 1, "hello world (English)"),
23          tempbuffer);
24    );

25    catclose(catalog);

26    exit(0);
27 ;

```

line 1	Needs to be included to use standard routines like printf(S) .
line 2	Needs to be included to use time-related functions and data types.
line 3	Needs to be included to use the catalog routines.
lines 4-5	Needs to be included to use the LC_* and LANG environment variables.
line 6	Defines the maximum size of a temporary buffer.
line 9	Defines a variable to be of a type that identifies a catalog (referred to as a catalog descriptor).
line 10	Defines a variable to store the time.
line 11	Defines a temporary buffer of size TEMPBUF_SIZE to store the formatted string corresponding to the time.
line 12	Needs to be included so that the local environment variable LANG is read.
line 13	Checks to see if a catalog with the <i>name</i> argument matching "hello" can be opened. If successful, this message catalog will now be referenced by the catalog variable. If the program is to use the LC_MESSAGES environment variable, use NL_CAT_LOCALE as the second argument to the catopen(S) call instead of the 0 shown in the example.
lines 14-16	Prints out error message upon failure.
line 17	Sets the clock variable to UNIX operating system Epoch time.
lines 18-20	Formats the time according to the format in the message catalog designated by the variable catalog . (The final format specification is a default if there is an error in reading the message catalog.)
lines 21-24	Retrieves message number 1 of set number 1 in the message catalog designated by the variable catalog . (The final string is a default if there is an error in reading the message catalog.) The retrieved message is printed and followed by the preformatted time string stored in tempbuffer .
line 25	Closes the message catalog identified by the variable catalog .

Combining the message catalog and the application program

This section describes how to compile the message catalog and how to locate it so that the application can use it.

Example 3-4 Setup program

```

1 set -x
2 NLSPATH=./%l/%N
3 export NLSPATH

4 gencat english/hello.cat english/hello.text
5 gencat french/hello.cat french/hello.text

6 cc hello.c -o hello
7 set +x

```

- | | |
|-----------|---|
| line 2 | Defines the NLSPATH variable which contains the path where the message catalogs are contained. The “%l” extracts the string defined in the LANG environment variable and the “%N” extracts the string from the <i>name</i> argument from the catopen(S) routine. If the value in the LANG variable is “english”, the message catalog file in our example would be <i>.english/hello.cat</i> |
| line 3 | Exports this definition so that the NLSPATH variable is defined permanently in this shell session. |
| lines 4-5 | Creates the <i>.cat</i> files for both the French and English message. The source files are compiled into binary files understood by the cataloging routines. |
| line 6 | Compiles the source for the main application program using the message catalogs. |

Running the application program

Now that the message catalog and the program have been compiled, the program is ready to be executed. Before continuing, you should ensure that the English and French *hello.cat* files are in the *./english* and *./french* subdirectories respectively.

Next at the command line prompt, enter:

```
1 $ ./setup
2 $ LANG=english; export LANG
3 $ ./hello
```

- | | |
|--------|--|
| line 1 | Runs the <i>setup</i> program that compiles the message catalog and the program. |
| line 2 | Sets the environment variable LANG ¹ to “english” and then it exports it to the environment. |
| line 3 | Runs the actual program, this line will produce output that looks similar to this: |

Hello everyone. It's Fri Oct 13 01:03:52 1973

Next the following two lines should be entered:

```
1 $ LANG=french; export LANG
2 $ ./hello
```

- | | |
|--------|--|
| line 1 | Sets the environment variable LANG to “french” and then it exports it to the environment. |
| line 2 | Again, the program is run, except that now the following output is printed instead: |

Bonjour tout le monde. C'est ven 13 oct 01h03 1973

1. Note that though XPG4 has suggested a certain format to define a particular locale, implementations are free to define their own format. Therefore, though it is acceptable to set **LANG** to “english” or “french”, these definitions do not match the XPG4 convention.

Chapter 4

Using trusted facilities

Because no computer system is completely free from risk, we refer to systems as “trusted” rather than “secure.” A trusted system achieves a greater level of control over access to information by providing mechanisms to detect and possibly prevent unauthorized access, along with additional means to confirm that these mechanisms are functioning properly. The C2 level of trust means that the system meets specific criteria in its security policy, accountability, assurance, testing, and documentation.

All UNIX operating systems provide some measure of security by allowing users have passwords to log on and offering the ability to specify access permissions for the owner, group, and other for each file or directory. SCO UNIX System V/386 Release 3.2 maintains compatibility with existing UNIX system mechanisms and provides enhanced security features and trusted facilities.

The examples in this chapter illustrate how to use trusted facilities to create a protected subsystem, which is an application that uses the trusted facilities to control access permission to its executable and data files.

Consider the following when deciding whether to implement your application using trusted facilities:

- Virtually all UNIX operating systems support the standard facilities, so programs that use the standard facilities are more portable.
- Trusted facilities provide a much greater level of security and are recommended for applications that manipulate data requiring protection from unauthorized disclosure.

Using protected subsystems

A protected subsystem is a collection of files, devices, and commands that protect a set of resources and perform security tasks.

The C code for a protected subsystem has the following characteristics:

- SecureWare must be defined in all programs that use the trusted software. Use the **-DSecureWare** option from CFLAGS on the compile line, or make sure the first line in the code is:

```
#define SecureWare
```

- Programs in a protected subsystem must have the following lines to reference required header files:

```
#include <sys/security.h>      /* kernel data structures*/
                               /* used for security functions*/
#include <sys/audit.h>         /* audit data structures */
#include <prot.h>              /* data structures used by libprot*/
```

- The first action in the **main()** routine must be **set_auth_parameters** as defined in the **identity(S)** manual page. Line 19 of “subsys.c example” (page 66), is an example of using **set_auth_parameters**.
- The program must be linked with the **-lprot** flag. An example link command is **cc myprogram.c -lprot**.

Subsystems are described in detail in the **subsystem(M)** manual pages.

Guidelines for using trusted facilities

Follow these guidelines when you use trusted facilities:

- Consult your security administrator before running programs that write sensitive system files, need super-user privilege, or manipulate kernel data structures.
- Be careful to use appropriate environment variables and data files. Set up a program's environment and signal handling dispositions to ensure that a user cannot make the program do something unintended; the program should catch **SIGINT**, **SIGQUIT**, and so on. This is essential for programs that execute other programs using the **exec(S)**, **system(S)**, and **popen(S)** commands. SUID programs should always set up the **IFS**, **PATH**, and **SHELL** environment variables.
- Use a restricted umask, such as 077, for programs that create private files.
- Use full (absolute) pathnames when executing program files from the shell.

- Protect programs from general readership.
- Setuid programs that run other programs are vulnerable; some can be penetrated easily. When passing control to a sub-program, use `setuid(S)` and `setgid(S)` to restore the original permissions. Make sure that all files needed by only the parent process are closed before invoking the child program. In general, use these privileges for the shortest amount of time possible. For example, if you need privilege to open a file, use:

```
main
    get enhanced priv
    open file
    drop enhanced priv
    write file
    ...
```

- Recall that the `access(S)` system call and the `eaccess(S)` system call are different. Use the appropriate one.

In most cases, use `eaccess(S)`, which checks against effective UID and GID. Use `access(S)`, which checks against the real UID and GID, when you are running a `setuid(S)` or `setgid(S)` binary.

- Catch and handle all expected signals. It is important to undo any partially done actions that might leave sensitive files in an inconsistent state.
- Leave things in a sensible state, in all error cases. Ensure that relevant exception conditions are handled appropriately.
- Handle concurrency. If two users are likely to run a program concurrently, make sure that any files to be modified are locked during the modification.
- Handle errors in file and user input formats. It is possible for any file to be corrupted, and errors that cause your program to fail or leave files or permissions in an inconsistent state invite penetrators. To make sure your system security is not compromised, check your return codes, and investigate and fix errors or inconsistencies. Be particularly conscious of overflowing buffers when reading input, which can cause unpredictable and potentially harmful behavior.

Programming examples

The rest of this chapter gives annotated code examples to illustrate how to program in a secure environment. Three code examples are given:

- | | |
|-----------------|--|
| subsys.c | a program similar to <code>auths(C)</code> , but it lists a user's subsystem authorizations rather than kernel authorizations |
| prwarn.c | a program that could be run from a user's <code>.login</code> or <code>.profile</code> file; it warns users when their passwords are about to expire |

loge.c a program similar to **su(C)**; it allows users to run certain programs as the super user without knowing the *root* password.

Throughout this chapter, these code examples use the basic trusted system concepts: TCB (Trusted Computing Base) and DAC (Discretionary Access Control). The TCB consists of the UNIX kernel and the trusted utilities that reference and maintain relevant security data. The TCB implements the system's security policy; at the C2 level, this consists of DAC. DAC determines whether a user or a user's process has access to the information they want.

For more information about the basic concepts of a trusted system, refer to the *NCSC Trusted Computer Systems Evaluation Criteria (TCSEC)*, the *DoD Password Management Guidelines*, and SCO's *System Administrator's Guide*.

subsys.c example

The **subsys.c** program is similar to **auths(C)**, but it lists a user's subsystem authorizations rather than kernel authorizations. The **subsys** command takes one argument, a user name. If no name is specified, **subsys** works on the effective user ID. It prints out the name of the account, the real name of the person assigned to the account, and the subsystem authorizations (both primary and secondary) for that account. This program can be run only by the owner of the account or the accounts administrator (in other words, a user with the **auth** subsystem authorization).

This program demonstrates how to:

- use **set_auth_parameters(S)** to initialize the security features
- use **getprpwnam(S)** to get a Protected Password database entry
- audit authentication problems with **audit_auth_entry(S)**
- check subsystem authorizations with **authorized_user(S)**
- audit database problems with **audit_security_failure(S)**
- use the **pr_passwd** structure with the system's default settings

Any program that queries the Protected Password database must be executed with the effective group ID (EUID) of **auth**. To install this program, make the binary **setgid auth**. For example, use the following commands:

```
chgrp auth subsys
chmod 2111 subsys
```

To compile this program, use the following command:

```
cc subsys.c -lprot -lx -o subsys
```

Example 4-1 subsys.c (Part 1/2)

```

1 #define SecureWare          /* enable security features      */
2 #include <sys/types.h>      /* system pseudo-types   */
3 #include <stdio.h>          /* standard I/O definitions */
4 #include <pwd.h>            /* /etc/passwd definitions */
5 #include <string.h>         /* string library definitions */
6 #include <sys/security.h>   /* various security definitions */
7 #include <sys/audit.h>     /* audit definitions      */
8 #include <prot.h>          /* database definitions    */

9 main(
10     int argc,
11     char *argv[]
12 ) {
13     struct pr_passwd *pr;    /* Protected Password entry */
14     struct passwd *pwd;      /* /etc/passwd entry        */
15     mask_t *pmask;          /* subsys authorization vector */
16     char *str;
17     int n, num;
18     int printed;

19     set_auth_parameters(argc, argv);

20     if (argc >= 2)
21         pwd = getpwnam(argv[1]);
22     else
23         pwd = getpwuid(starting_euid());
24     if (pwd == (struct passwd *)0) {
25         (void) fprintf(stderr, "%s: No such user.\n", command_name);
26         exit(2);
27         /* NOTREACHED */
28     }

29     if (pwd->pw_uid != starting_luid() && ! authorized_user("auth")) {
30         (void) fprintf(stderr, "%s: Not authorized.\n", command_name);
31         exit(3);
32         /* NOTREACHED */
33     }

```

- line 1 **SecureWare** must be defined in programs that use the trusted software. If it is not defined on the compile line, the `#define SecureWare` statement is required. This allows the security-related definitions in the header files to be interpreted correctly.
- line 4 *pwd.h* defines the **passwd** structure. Trusted software uses this information to identify the user, as in line 24. This file must be included to use such routines as **getpwnam(S)** (line 21) and **getpwuid(S)** (line 23).
- line 5 *string.h* is required for the **strchr(S)** call in line 35. The manual page for each routine defines which `#include` lines are required when calling that routine.
- lines 6-8 These lines contain header files that define the trusted software.
- line 19 This line initializes the authentication routines. This must be the first action in **main()**. The two arguments, count and vector, call the program. The **identity(S)** manual page describes **set_auth_parameters()**.
- lines 20-28 These lines get the */etc/passwd* entry for the account named on the command line. If no name is specified, use the current EUID (Effective User Identification). If there is no entry in */etc/passwd* (in other words, the account does not exist), then exit with an error message.
- Note that it is not a security relevant event when the program reports that an account does not exist, because you can determine this by looking at the */etc/passwd* file which can be read by all users.
- lines 29-33 These lines check that the user is permitted to see the authorizations for the account. Only the account owner (the UID matches the LUID) or the accounts administrator (a user who has **auth** subsystem authorization) can check the authorizations for the account. An unauthorized user does not need to know the authorizations of another user.

Example 4-2 subsys.c (Part 2/2)

```

34     (void) printf("User name: %s\n", pwd->pw_name);
35     if ((str = strchr(pwd->pw_gecos, ',')) != (char *)0)
36         *str = '\0';
37     (void) printf("Real name: %s\n", pwd->pw_gecos);
38     (void) printf("Subsystems:");

39     pmask = (mask_t *)0;
40     if ((pr = getprpwnam(pwd->pw_name)) == (struct pr_passwd *)0) {
41         audit_auth_entry(pwd->pw_name, OT_PRPWD,
42             "User missing from Protected Password database"
43         );
44         str = "UNKNOWN (no Protected Password entry)";
45     }
46     else if (build_cmd_priv() || (num = total_auths()) < 0) {
47         audit_security_failure(
48             OT_SUBSYS, 1L, 0L, "Subsystem names", "Not listed"
49         );
50         str = "UNKNOWN (cannot build list of subsystem names)";
51     }

52     else if (pr->uflg.fg_cprivs)
53         pmask = pr->uflg.fg_cprivs;
54     else if (pr->sflg.fg_cprivs)
55         pmask = pr->sflg.fg_cprivs;
56     else
57         str = "NONE";
58     printed=0;

59     if (pmask == (mask_t *)0)
60         (void) printf(" %s", str);
61     else {
62         for (n = 0; n < num; n++)
63             if (ISBITSET(pmask, cmd_priv[n].value)) {
64                 printed=1;
65                 (void) printf(" %s", cmd_priv[n].name);
66             }
67     }
68     if (!printed)
69         (void) printf(" NONE");
70     putchar('\n');

71     (void) fflush(stdout);
72     exit(ferror(stdout));
73     /* NOTREACHED */
74 }

```

- lines 34-38 These lines report the name of the account and the person's real name (which is assumed to be the part of the `pw_gecos` field in the `/etc/passwd` file that precedes the first comma).
- lines 39-45 These lines get the subsystem authorization vector for the account. Because the account exists in the `/etc/passwd` file, it must have a Protected Password entry. If it does not, it is an auditable problem. (Note that this may not be true if the account is defined via client-mode NIS, but it will be hidden from the calling program.)
- lines 41-43 The `audit_auth_entry(S)` call produces this type of an audit record. The first argument is the name of the user from the `/etc/passwd` file; the second argument identifies the type of problem (`OT_PRPWD` is the Protected Password database; see the `audit.h` header file for a list of valid *type* flags); the third argument is a literal string that is recorded in the audit database. See `authaudit(S)` for a list of routines that record other types of audit records.
- lines 46-51 These lines obtain the names of the installed subsystems; if none are listed, audit the failure.
- lines 47-49 The first argument to `audit_security_failure` defines the object in which the problem was detected; `OT_SUBSYS` indicates the Subsystem database. See the `audit.h` header file for a list of valid values. The second and third arguments record the expected and current values to help in diagnosing the problem; the fourth and fifth arguments identify additional information about the problem. In these sample programs, the messages are hard-coded in English. If you anticipate needing to localize your application, implement these messages (lines 47-51) using a message catalog so the messages print in the chosen language of the user.
- lines 52-58 If the `fg_cprivs` flag is set in the `pr_passwd` structure, the account has a private list of subsystem authorizations. If not, the system default applies. If the system default list does not exist, there are no default authorizations; in this specific case, this is not considered an error. However, in most cases, this would be considered an error that should be audited. If there are no default authorizations, assign a restrictive value for the authorizations.
- lines 59-70 These lines list the subsystem authorizations for the account.

prwarn.c example

The **prwarn.c** program warns users when their passwords are about to expire. The **prwarn** command takes three optional arguments. The arguments specify the number of days before a password expires that a warning is given, the number of hours and minutes between warnings, and the user name(s) who receives the warning. The logged-in user must be authorized to change the passwords of all specified users.

The program is usually run from a user's *.profile* or *.login* file, in which case no user names are specified and the warning applies to this user. The time of the warning is saved as the modification time of the *\$HOME/prwarn_time* file.

This program demonstrates how to:

- use **set_auth_parameters(S)** to initialize the security features
- check subsystem authorizations with **authorized_user(S)**
- use **getprpwnam(S)** to get a Protected Password database entry
- audit authentication database problems with **audit_auth_entry(S)**
- compute and comparing password lifetime parameters
- determine who is authorized to change a user's password
- use the **pr_passwd** structure, given the system's default settings

Any program that queries the Protected Password database must be executed with the effective group ID (EUID) of **auth**. To install this program, make the binary **setgid auth**. For example, use the following commands:

```
chgrp auth prwarn
chmod 2111 prwarn
```

Compile this program with the following command:

```
cc prwarn.c -lprot -lx -o prwarn
```

The **-lprot** option searches the *prot* library that contains the trusted routines.

This program is structured so that the subroutines are defined in front of the **main** routine, which begins in line 291.

Example 4-3 prwarn.c (Part 1/13)

```

1 #define SecureWare          /* enable security features      */
2 #include <sys/types.h>      /* system pseudo-types   */
3 #include <stdio.h>          /* standard I/O definitions */
4 #include <string.h>         /* string library definitions */
5 #include <stdarg.h>         /* variable-len argument lists */
6 #include <unistd.h>         /* access(S) definitions   */
7 #include <sys/security.h>   /* various security definitions */
8 #include <sys/audit.h>      /* audit definitions       */
9 #include <prot.h>           /* database definitions    */
10 #include <pwd.h>            /* /etc/passwd definitions */
11 #include <time.h>           /* localtime(S) definitions */
12 #include <utime.h>          /* utime(S) definitions    */
13 #include <sys/stat.h>       /* stat(S) definitions     */
14 extern int      defopen(char *);
15 extern char     *defread(char *);

```

- line 1 **SecureWare** must be defined in programs that use the trusted software. If it is not defined on the compile line, the `#define SecureWare` statement is required. This allows the security-related definitions in the header files to be interpreted correctly.
- line 5 This header file is necessary because the code defines functions with variable length argument lists.
- lines 7-9 These lines include the three header files that define the trusted facilities.
- line 10 *pwd.h* defines the **passwd** structure. Trusted software uses this information to identify the user.
- lines 11-12 These header files define time conventions used to calculate the time between warnings and the time until the password expires.
- lines 14-15 **defopen** and **defread** are used to read default program options. See **defopen(S)** for more information.

Example 4-4 prwarn.c (Part 2/13)

```

16 #define TRUE          1
17 #define FALSE         0
18 #define ARBSTRSIZ     300      /* ARbitrary STRing SIZE*/
19 #define SECONDS_PER_HOUR (60L * 60L)
20 #define SECONDS_PER_DAY (24L * SECONDS_PER_HOUR)
21 time_t    now;
22 int       is_administrator;      /* TRUE if LUID is Accounts Administrator */

23 char      prwarn_file[]        = ".prwarn_time";
24 char      default_file[]       = "/etc/default/prwarn";

25 time_t    silence = 6 * SECONDS_PER_HOUR;
26 int       always = FALSE;
27 int
28 set_remind_time(
29     char *str
30 ) {
31     int hhmm, hour, mins;

32     if (strcmp(str, "always") == 0) {
33         always = TRUE;
34         return (0);
35     }
36     hhmm = atoi(str);
37     if (strlen(str) < 2) {
38         hour = hhmm;
39         mins = 0;
40     }
41     else {
42         hour = hhmm / 100;
43         mins = hhmm % 100;
44     }
45     if (hour < 0 || mins < 0 || 59 < mins)
46         return (-1);
47     silence = (((time_t)hour * 60L) + (time_t)mins) * 60L;
48     always = FALSE;
49     return (0);
50 }

```

lines 16-20	These global <code>#define</code> statements are shown here for simplicity, but in an application with several programs it might make sense to put these definitions in an application header file that would use <code>#include</code> statements in all programs for the application.
lines 16-17	The literals TRUE and FALSE are used to improve readability.
line 18	ARBSTRSIZ is used in lines 83 and 298.
lines 19-20	The time resolutions for this program are in seconds. Defining constants for these time spans increases the readability of the code.
line 21	This line provides storage for the current date and time. The time_t type is defined in the <code>/usr/include/sys/types.h</code> file.
line 22	This line is TRUE if LUID (logged in user identification) is accounts administrator.
line 23	This line specifies the name of the file used to track the previous warning time for this user.
line 24	This line specifies the name of the file used to store system defaults for the program.
line 25	The default time between reports is six hours. This program uses the second as the time measurement, and determines the time of the report by multiplying the number of hours by the constant defined in line 19. Lines 47-49 override this default value.
line 26	This line defines a global flag.
lines 27-50	These lines parse the <code><hhmm></code> string using standard string comparison functions.
lines 45-46	These lines check that the value of hour and mins is valid. If no return, a -1 indicates an error. The program does not exit at this point, although this is a fatal error. This error can only result if the user made a typing error when issuing the command. When this routine is called (line 317), an error return causes it to call the usage routine defined in lines 272-291. usage prints an error message, a reminder of the correct syntax of the command, and then exits.

Example 4-5 prwarn.c (Part 3/13)

```

51 time_t    forced    = 7 * SECONDS_PER_DAY;        /* -d (time before expires) */
52 int       nolimit    = FALSE;                      /* TRUE if no -d cutoff time */

53 int
54 set_time_before(
55     char *str
56 ) {
57     int days;

58     if (strncmp(str, "inf", 3) == 0) {
59         nolimit = TRUE;
60         return (0);
61     }
62     if ((days = atoi(str)) < 0)
63         return (-1);

64     forced = (time_t)days * SECONDS_PER_DAY;
65     nolimit = FALSE;
66     return (0);
67 }

68 int
69 days_in_year(
70     register int yr
71 ) {
72     if ((yr % 4) == 0 && (yr % 100) != 0) || (yr % 400) == 0)
73         return (366);
74     return (365);
75 }

```

lines 53-67 These lines parse the *<days>* string.

lines 68-75 These lines determine the number of days in a specified Julian year.¹

-
1. A year is a leap year if it is divisible by 4 but not by 100, except that years divisible by 400 are leap years. The “less three days every four hundred years” rule is the difference between the old Gregorian and modern Julian 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.

Example 4-6 prwarn.c (Part 4/13)

```
76 char *
77 when(
78     time_t then
79 ) {
80     struct tm today, other;
81     long days;

82     static char buf[ARBSTRSIZ];

83     other = *localtime(&then);
84     today = *localtime(&now);
85     if (other.tm_year == today.tm_year+1) {           /* next year */
86         other.tm_yday += days_in_year(today.tm_year);
87         other.tm_year = today.tm_year;
88     }
89     else if (other.tm_year == today.tm_year-1) {      /* last year */
90         today.tm_yday += days_in_year(other.tm_year);
91         today.tm_year = other.tm_year;
92     }
93     if (other.tm_year == today.tm_year) {
94         if (other.tm_yday == today.tm_yday-1)
95             return ("yesterday");
96         if (other.tm_yday == today.tm_yday)
97             return ("today");
98         if (other.tm_yday == today.tm_yday+1)
99             return ("tomorrow");
100     }
101     if (then > now) {
102         days = (then - now) / SECONDS_PER_DAY;
103         (void) sprintf(buf, "in %ld days", days);
104     }
105     else {
106         days = (now - then) / SECONDS_PER_DAY;
107         (void) sprintf(buf, "%ld days ago", days);
108     }
109     return (buf);
110 }
```

lines 76-110 These lines calculate the relation between the specified date and today.

Example 4-7 prwarn.c (Part 5/13)

```

111 time_t
112 last_reported(
113     struct passwd *pwd
114 ) {
115     struct stat stbuf;
116     time_t last;

117     if (chdir("/"))
118         perror("Cannot change current working directory to /");
119     else if (chdir(pwd->pw_dir) == 0 && stat(prwarn_file, &stbuf) == 0)
120         return (stbuf.st_mtime);

121     return (0);                /* The Epoch: Jan 1st 1970 00:00 GMT */
122 }

123 void
124 just_reported(
125     struct passwd *pwd
126 ) {
127     struct utimbuf utbuf;

128     if (eaccess(prwarn_file, F_OK) && close(creat(prwarn_file, 0600)) == 0)
129         (void) chown(prwarn_file, pwd->pw_uid, pwd->pw_gid);
130     utbuf.actime = now;
131     utbuf.modtime = now;
132     (void) utime(prwarn_file, &utbuf);
133 }

```

- lines 111-122 These lines determine the date and time that the user was last warned that the password is about to expire. The modification time of the *\$HOME/prwarn_time* file is used as the last time the user was notified.
- lines 117-118 The home directory is treated relative to */*, even if the listing in the */etc/passwd* file is not absolute.
- lines 119-120 This code uses the **stat(S)** system call to return statistics on the file; the **st_mtime** value returned is the last modification time stored in the file's inode.
- lines 123-133 These lines remember that a pending expiration was just reported. This code assumes that the current working directory is the user's home directory as set by the **last_reported** routine in lines 111-122.

Example 4-8 prwarn.c (Part 6/13)

```
134 time_t
135 expires_at(
136     register struct pr_passwd *pr_pw
137 ) {
138     if (pr_pw->uflg.fg_expire)
139         return (pr_pw->ufld.fd_expire);
140     else if (pr_pw->sflg.fg_expire)
141         return (pr_pw->sfld.fd_expire);

142     audit_security_failure(
143         OT_DFLT_CNTL, -1L, 0L, AUTH_DEFAULT,
144         "No system default password expiration time"
145     );
146     return (0);
147 }

148 time_t
149 minchange_is(
150     register struct pr_passwd *pr_pw
151 ) {
152     if (pr_pw->uflg.fg_min)
153         return (pr_pw->ufld.fd_min);
154     else if (pr_pw->sflg.fg_min)
155         return (pr_pw->sfld.fd_min);

156     audit_security_failure(
157         OT_DFLT_CNTL, -1L, 0L, AUTH_DEFAULT,
158         "No system default minimum password change time"
159     );
160     return (0);
161 }
```

lines 134-147 This code calculates the date and time after which a password must be changed. "0" is interpreted as infinity; in other words, the password has no expiration date.

lines 148-161 This code calculates the time that must elapse between password changes.

Example 4-9 prwarn.c (Part 7/13)

```

162 time_t
163 lives_until(
164     register struct pr_passwd *pr_pw
165 ) {
166     if (pr_pw->uflg.fg_lifetime)
167         return (pr_pw->ufld.fd_lifetime);
168     else if (pr_pw->sflg.fg_lifetime)
169         return (pr_pw->sfld.fd_lifetime);

170     audit_security_failure(
171         OT_DFLT_CNTL, -1L, 0L, AUTH_DEFAULT,
172         "No system default password lifetime"
173     );
174     return (0);
175 }

176 time_t
177 last_changed_at(
178     register struct pr_passwd *pr_pw
179 ) {
180     if (pr_pw->uflg.fg_schange)
181         return (pr_pw->ufld.fd_schange);

182     return (0);
183 }

```

-
- lines 162-175 This code calculates the date or time after which the password cannot be used.
- lines 176-183 This code calculates the date or time at which the password was last changed.

Example 4-10 prwarn.c (Part 8/13)

```
184 enum password_change {
185     CanBeChanged,
186     TooEarly,
187     Procrastinated
188 } changeable{
189     register struct pr_passwd *pr_pw
190 } {
191     time_t lastchg, life;

192     if ((lastchg = last_changed_at(pr_pw)) == 0)
193         return (CanBeChanged);

194     if (lastchg + minchange_is(pr_pw) > now)
195         return (TooEarly);

196     if ((life = lives_until(pr_pw)) != 0 && lastchg + life < now)
197         return (Procrastinated);

198     return (CanBeChanged);
199 }

200 int
201 authorized_to_change(
202     register struct pr_passwd *pr_pw
203 ) {
204     if (is_administrator)
205         return (TRUE);
206     else if (pr_pw->uflg.fg_pswduser)
207         return (pr_pw->ufld.fd_pswduser == starting_luid());
208     else if (pr_pw->sflg.fg_pswduser)
209         return (pr_pw->sfld.fd_pswduser == starting_luid());
210     else if (pr_pw->uflg.fg_uid)
211         return (pr_pw->ufld.fd_uid == starting_luid());
212     return (FALSE);
213 }
```

- lines 184-199 These lines determine whether the password can be changed now, and if not, why not. **CanBeChanged** means that the password can be changed now. **TooEarly** means that the password cannot be changed at this time because not enough time has elapsed since it was last changed. **Procrastinated** means that the password should have been changed before now; the password is expired but the lifetime is not up. **CanBeChanged** is referenced in lines 198 and 259; **TooEarly** is referenced in line 195; **Procrastinated** is referenced in lines 197 and 262.
- lines 192-193 If the password was never set, it is legal to set it.
- lines 194-195 If the time since the last change was less than the “minimum time change” calculated in lines 148-161, the password cannot be changed.
- lines 196-197 If the password has a finite lifetime and was last changed earlier than the date calculated in lines 162-169, the password cannot be changed (and the user cannot log in).
- line 198 Otherwise, the password can be changed. This code makes no provision for the possibility that the user has retired or left the company.
- lines 200-213 This code checks whether the current user (LUID) is authorized to change this user’s password. An unauthorized user has no reason to know when another user’s password expires. The user running this program is authorized to know when a password expires if, and only if, the user running this program can change the password. This routine is called in line 231.
- lines 204-205 If the LUID has the **auth** subsystem authorization, the password may be changed.
- lines 206-207 If the user has a **pswduser**, the LUID can change the password if it is the user’s **pswduser**.
- lines 208-209 If the system has a default **pswduser**, the password can be changed if, and only if, the LUID is the system default **pswduser**. Note that this condition is met only if the user does not have a **pswduser**.
- lines 210-211 Users can change their own passwords.
- line 212 If none of the above conditions are met, the LUID running the program is not authorized to change this password. It is possible to prevent a user from ever changing the password.

Example 4-11 prwarn.c (Part 9/13)

```

214 int
215 report(
216     struct passwd *pwd,
217     int do_update
218 ) {
219     struct pr_passwd *pr_pw;
220     time_t expire, lastchg, nextchg;

221     if ((pr_pw = getprpwnam(pwd->pw_name)) == (struct pr_passwd *)0) {
222         audit_auth_entry(pwd->pw_name, OT_PRPWD,
223             "User missing from Protected Password database"
224         );
225         (void) fprintf(stderr,
226             "%s: Missing Protected Password entry: %s\n",
227             command_name, pwd->pw_name
228         );
229         return (1);
230     }

231     if ( ! authorized_to_change(pr_pw) ) {
232         (void) fprintf(stderr,
233             "%s: Sorry, you are not authorized to change %s's password.\n",
234             command_name, pwd->pw_name
235         );
236         return (1);
237     }

```

lines 214-271 These lines report to the user if the password expires soon.

lines 221-230 The user has an entry in the */etc/passwd* file, so without the Protected Password entry, you can audit the problem.

lines 231-237 If the user is not authorized to change the password, print an error message. The **authorized_to_change** subroutine is defined in lines 200-213.

Example 4-12 prwarn.c (Part 10/13)

```

238         if ( ! always && now - last_reported(pwd) < silence)
239             return (0);

240         expire = expires_at(pr_pw);
241         lastchg = last_changed_at(pr_pw);
242         nextchg = lastchg + expire;
243         if (lastchg == 0 || nolimit || (expire && nextchg - forced <= now)) {
244             if (do_update)
245                 (void) fputs("Your password", stdout);
246             else
247                 (void) printf("The password for %s", pwd->pw_name);
248             if (expire == 0)
249                 (void) fputs(" never expires", stdout);
250             else if (lastchg == 0)
251                 (void) fputs(" has never been set", stdout);
252             else
253                 (void) printf(" expire%c %s at %s",
254                             (nextchg < now) ? 'd' : 's',
255                             when(nextchg),
256                             nl_cxtime(&nextchg, ""));
257             );
258             switch (changeable(pr_pw)) {
259             case CanBeChanged:
260                 (void) fputs(",\n\tand can be changed", stdout);
261                 break;
262             case Procrastinated:
263                 (void) fputs(",\n\tand is now dead", stdout);
264                 break;
265             }
266             (void) fputs(".\n", stdout);
267             if (do_update)
268                 just_reported(pwd);
269         }
270         return (0);
271 ]

```

lines 238-239	The user sets the frequency at which to be reminded that the password is about to expire. If that time period is greater than the lapse since a warning was issued, do not issue a warning and do not update the time of last warning.
lines 240-271	Otherwise, issue a warning that the password must be reset soon, and update the time of the last warning.
line 255	Lines 77-110 define the when subroutine. The subroutine returns a human-readable string representing a number of day in the future or past.
line 259	Line 185 defines the CanBeChanged case.
line 262	Line 187 defines the Procrastinated case.
line 268	Lines 123-133 defines the just_reported routine. The routine changes the date on the users' <i>.prwarn_time</i> file to record when they were last warned.

Example 4-13 prwarn.c (Part 11/13)

```

272 void
273 usage(
274     char *fmt,
275     ...
276 ) {
277     va_list args;

278     if (fmt != (char *)0) {
279         (void) fprintf(stderr, "%s: ", command_name);

280         va_start(args, fmt);
281         (void) vfprintf(stderr, fmt, args);
282         va_end(args);

283         putc('\n', stderr);
284     }
285     (void) fprintf(stderr,
286         "Usage: %s [ -d days ] [ -t hh:mm ] [ users ]...\n",
287         command_name
288     );
289     exit(2);
290     /* NOTREACHED */
291 }

```

lines 272-291 This subroutine prints the usage message and exits the program if the user issues the command incorrectly. It is called in lines 313, 317, and 321.

Example 4-14 prwarn.c (Part 12/13)

```
292 main(
293     int argc,
294     char *argv[]
295 ) {
296     int c, nerrs;
297     char *str, temp[ARBSTRSIZ];
298     struct passwd *pwd;

299     set_auth_parameters(argc, argv);

300     (void) time(&now);
301     is_administrator = authorized_user("auth");

302     if (defopen(default_file) == 0) {
303         if ((str = defread("DAYS_BEFORE=")) != (char *)0)
304             (void) set_time_before(str);
305         if ((str = defread("REMIND_TIME=")) != (char *)0)
306             (void) set_remind_time(str);
307         (void) defopen((char *)0);
308     }

309     while ((c = getopt(argc, argv, "d:t:") != EOF) {
310         switch (c) {
311             case 'd':
312                 if (set_time_before(optarg))
313                     usage("Invalid number of days: %s", optarg);
314                 break;
315             case 't':
316                 if (set_remind_time(optarg))
317                     usage("Invalid time interval: %s", optarg);
318                 break;
319             case '?':
320             default:
321                 usage((char *)0);
322             }
323     }
```

line 292	This line begins the main() routine. Thus far, everything has been subroutines that are called as functions in main() .
line 299	set_auth_parameters(S) must be the first function called in main() .
lines 300-301	These lines initialize the current date and time, and determine whether or not the logged-in user is an accounts administrator.
lines 302-308	If there are any legal system-wide defaults, use them rather than the program's built-in defaults (lines 25-26 and 51-52).
lines 309-323	If the user supplied explicit values on the command line, check that they are legal and, if so, use these values rather than the program or system defaults.
lines 324-349	If the command line specifies additional users, report on pending expirations for those users but do not update the time-of-last-report. Otherwise, report only on the logged-in user, updating the time-of-last-report as appropriate.
lines 324-336	This code converts the logged-in user's UID to the user's name and reports a pending expiration. If it is missing from the Protected Password database, it audits the fact and prints an error message.
lines 337-349	If the named user is not listed in the <i>/etc/passwd</i> file, then there is no user by that name. In this case, print an error message to the user's terminal.

Example 4-15 prwarn.c (Part 13/13)

```

324         if (optind >= argc) {
325             if ((pwd = getpwuid(starting_luid())) != (struct passwd *)0)
326                 exit(report(pwd, TRUE));
327             (void) sprintf(temp, "UID %d", starting_luid());
328             audit_auth_entry(temp, OT_PWD,
329                 "User missing from /etc/passwd"
330             );
331             (void) fprintf(stderr,
332                 "%s: Cannot determine your username!\n",
333                 command_name
334             );
335             exit(1);
336         }
337     else {
338         while (optind < argc) {
339             pwd = getpwnam(argv[optind]);
340             if (pwd != (struct passwd *)0)
341                 nerrs += report(pwd, FALSE);
342             else
343                 (void) printf("There is no user named %s.\n",
344                     argv[optind]
345                 );
346             optind++;
347         }
348         exit(nerrs != 0);
349     }
350     /* NOTREACHED */
351 }

```

line 324 This line checks **optind** from line 14. **optind** is the index in **argv** of the next argument, so if **optind** >= **argc**, the code has processed all of the command-line arguments.

lines 325-326 This code tries to get the Protected Password entry for the login user ID stored when **set_auth_parameters** was called. If it is successful, it calls the **report** routine (lines 214-271) to print a report about the current user. If it fails, it audits the fact and prints error messages.

log.e.c example

The **log.e.c** program allows users who have the appropriate authorization to execute a command with super-user privileges. Users must supply a password for verification. They do not need to supply the super-user password. **log.e.c** can be used with any command whose executable file is in the */tcb/files/valhalla* directory. Each command can also define a secondary subsystem authorization.

The logged-in user (LUID) is authorized if they have either the “**wotan**” primary subsystem authorization or the secondary subsystem authorization defined for the command issued under **log.e.c**.

This program must be installed with both **setuid root** and **setgid wotan**.

This program demonstrates how to:

- create and use a Protected Subsystem
- use **set_auth_parameters(S)** to initialize the security facilities
- use **getprpwuid(S)** to get a Protected Password database entry
- audit authentication database problems with **audit_auth_entry(S)**
- check subsystem authorizations with **authorized_user(S)**
- use **getprfinam(S)** and the File Control database
- audit integrity problems with **audit_lax_file**
- prompt for a clear-text password with **getpasswd(S)**
- check against the encrypted password with **bigcrypt(S)**
- audit a system administrator’s usage of a password with **sa_audit_subsystem** (see **audit_subsystem(S)**)
- create a file with the attributes listed in the File Control database by using **create_file_securely(S)**
- audit unexpected or unlikely failures with **audit_no_resource(S)**
- use the **pr_passwd** and **pr_file** structures, taking into account the system’s default settings

Compile and install this program with the following command:

```
cc log.e.c -lprot -lx -o loge
chown root loge
chgrp wotan loge
chmod 6111 loge
```

Example 4-16 loge.c (Part 1/17)

```

1 #define SecureWare          /* enable security features      */
2 #include <sys/types.h>      /* system pseudo-types    */
3 #include <errno.h>          /* system error definitions */
4 #include <stdio.h>          /* standard I/O definitions */
5 #include <ctype.h>          /* character classification */
6 #include <string.h>         /* string library definitions */
7 #include <stdarg.h>         /* variable-len argument lists */
8 #include <unistd.h>         /* access(S) definitions   */
9 #include <sys/stat.h>       /* stat(S) definitions     */
10 #include <time.h>           /* localtime() definitions  */
11 #include <sys/security.h>   /* various security definitions */
12 #include <sys/audit.h>     /* audit definitions        */
13 #include <prot.h>          /* database definitions     */

14 extern char      *ttyname(int), *defread(char *);
15 extern int       defopen(char *);

16 #define TRUE      1
17 #define FALSE     0

18 #define ARBSTRSIZ 300          /* ARBiTrary STRing SIze      */
19 #define MAXCMDLEN ARBSTRSIZ    /* MAXimum CoMmand name LENgth */

20 char      primary_subsystem[] = "wotan";
21 char      valhalla[]         = "/tcb/files/valhalla";
22 char      su_defaults[]      = "/etc/default/su";

23 void      syserror(int, char *, ...);

24 void      sulog(char *, int, char **);
25 void      failed(int, char *, char **);

```

Example 4-17 loge.c (Part 2/17)

```

26 #define TCB_ANY_TYPE      ' '           /* any type of file          */
27 #define TCB_FILE          'r'           /* regular file             */
28 #define TCB_DIRECTORY     'd'           /* directory                 */

29 int      check_TCB_path(char *, char, char *);
30 int      check_DAC_path(char *, char, char *);

31 enum DAC_status {
32     NotListed,           /* file not listed in File Control database */
33     NoSuchFile,          /* file does not exist                     */
34     WrongDAC,             /* existing file does not meet required DAC */
35     MatchesDAC,           /* existing file has exactly the required DAC */
36     ExceedsDAC            /* existing file grants less access than required */
37 } compare_DAC_attributes(char *, struct pr_file *);

38 main(
39     int argc,
40     char *argv[])
41 ) {
42     struct pr_passwd *pr_pw;           /* Protected Password entry */
43     char path[sizeof(valhalla) + 1 + MAXCMDLEN + 1];
44     char temp[sizeof(path) + ARBSTRSIZ];
45     char *clear, *cipher;              /* Clear-, cipher-text password */
46     int password_required;              /* TRUE if must have a password */
47     int has_password;                   /* TRUE if has a password      */
48     int is_proper;                       /* TRUE if knows password      */

```

Example 4-18 loge.c (Part 3/17)

```

49     set_auth_parameters(argc, argv);

50     if (argc < 2 || *argv[1] == ' ') {
51         (void) fprintf(stderr,
52             "Usage: %s cmd [ args ]...\n",
53             command_name
54         );
55         failed(2, (char *)0, &argv[1]);
56         /* NOTREACHED */
57     }
58     if (strchr(argv[1], '/') != (char *)0 || strlen(argv[1]) > MAXCMDLEN) {
59         (void) fprintf(stderr,
60             "%s: Not the name of a command: %s\n",
61             command_name, argv[1]
62         );
63         failed(1, (char *)0, &argv[1]);
64         /* NOTREACHED */
65     }

```

line 49 This line initializes the authentication routines. This must be the first function called in the **main()** routine. A side effect of this call is that the **umask** is changed to 077.

lines 50-65 The command must be issued with at least one argument that corresponds to the base name of an executable file in the */tcb/files/valhalla* directory. This program provides this information directly within the **main** routine rather than using a separate subroutine as the *prwarn.c* example did. Lines 154-164 define the **failed** subroutine.

Example 4-19 loge.c (Part 4/17)

```

66         if ( ! authorized_user(primary_subsystem) && ! authorized_user(argv[1])) {
67             (void) fprintf(stderr,
68                 "Sorry, not authorized to run %s.\n",
69                 argv[1]
70             );
71             failed(1, (char *)0, &argv[1]);
72             /* NOTREACHED */
73         }

74         (void) sprintf(path, "%s/%s", valhalla, argv[1]);
75         if (eaccess(path, X_OK)) {
76             syserror(errno, "%s is not an executable command", path);
77             failed(1, (char *)0, &argv[1]);
78             /* NOTREACHED */
79         }

```

lines 66-73 These lines check the permissions of the logged-in user. The user must either have the primary subsystem authorization (“**wotan**”) or must have the specific secondary subsystem authorization associated with the specified command. After this check is passed, we know that the LUID is authorized. Consequently, there is no need to disguise any failures or otherwise hide any information.

lines 74-79 These lines check that the specified command exists in the *valhalla* directory and that it is executable.

Example 4-20 loge.c (Part 5/17)

```

80         if (check_TCB_path(path, TCB_FILE, "restricted command")) {
81             (void) fprintf(stderr,
82                 "%s: The system's integrity may be compromised.\n",
83                 command_name
84             );
85             failed(1, (char *)0, &argv[1]);
86             /* NOTREACHED */
87         }

88         if ((pr_pw = getprpwuid(starting_luid())) == (struct pr_passwd *)0 ||
89             ! pr_pw->uflg.fg_name
90         ) {
91             (void) sprintf(temp, "UID %d", starting_luid());
92             audit_auth_entry(temp, OT_PRPWD,
93                 "User missing from Protected Password database"
94             );
95             (void) fprintf(stderr,
96                 "%s: Cannot obtain authentication data for %s",
97                 command_name, temp
98             );
99             failed(1, (char *)0, &argv[1]);
100            /* NOTREACHED */
101        }

102 #ifdef AUTH_U_NULLPW                                /* #define'd in 3.2v2.0      */
103         if (pr_pw->uflg.fg_nullpw)
104             password_required = ( ! pr_pw->uflg.fg_nullpw);
105         else if (pr_pw->sflg.fg_nullpw)
106             password_required = ( ! pr_pw->sflg.fg_nullpw);
107         else
108 #endif
109             password_required = TRUE;

110         if (pr_pw->uflg.fg_encrypt && *pr_pw->uflg.fg_encrypt) {
111             has_password = TRUE;
112             cipher = pr_pw->uflg.fg_encrypt;
113         }
114         else {
115             has_password = FALSE;
116             cipher = "";                                /* not a valid encrypted password */
117         }

```

lines 80-87	These lines check that the file conforms to the specification listed in the FileControl database.
lines 88-101	These lines get the subsystem authorization vector for the LUID. Because the user has logged in previously, there must be a Protected Password Entry; if not, this is an auditable problem. It also checks that the LUID has a name, because you need the name later (and all Protected Password entries must have names).
lines 102-117	These lines check whether the LUID needs a password. Before Release 3.2, Version 2.0 and SLS UNIX 223, passwords were always required.
line 112	Empty cipher text is the same as no password.

Example 4-21 loge.c (Part 6/17)

```

118     if (password_required || has_password) {
119         clear = getpasswd("Password:", AUTH_MAX_PASSWD_LENGTH);
120         is_proper = (strcmp(cipher, bigcrypt(clear, cipher)) == 0);
121         (void) getpasswd((char *)0, AUTH_MAX_PASSWD_LENGTH);
122         if (is_proper) {
123             (void) sprintf(temp,
124                 "Password correct - run %s as superuser",
125                 path
126             );
127             sa_audit_subsystem(ST_AUTH, "Check password", temp);
128         }
129         else {
130             (void) sprintf(temp,
131                 "Password incorrect - command %s not run",
132                 path
133             );
134             sa_audit_subsystem(ST_AUTH, "Check password", temp);
135             (void) fputs("Sorry.\n", stderr);
136             failed(1, pr_pw->uflld.fd_name, &argv[1]);
137             /* NOTREACHED */
138         }
139     }

140     sulog(pr_pw->uflld.fd_name, TRUE, &argv[1]);

141     endprpwent();           /* Close Protected Password database */
142     endprfient();           /* Close File Control database */

143     if (putenv("PATH=/bin:/etc:/usr/bin:/tcb/bin") || putenv("IFS= \t\n"))
144         audit_no_resource("environment", OT_MEMORY, "cannot change");
145     else if (setgid(starting_rgid()))
146         syserror(errno, "Cannot reset GID");
147     else {
148         (void) execv(path, &argv[1]);
149         syserror(errno, "Could not run %s", path);
150     }
151     exit(3);
152     /* NOTREACHED */
153 }

```

lines 118-140

These lines request the user to supply the password for the LUID and authenticate it. The “Orange Book” requires audit of “use of authentication mechanisms.”

bigcrypt() is a function that has the same parameters and function as the traditional **crypt()** library routine. In addition, it implements an extension to the **crypt** function to support longer passwords.²

The password encryption scheme on SCO UNIX system V/386, Release 3.2, is designed to have compatibility with earlier UNIX system and XENIX password encryption, and to have the ability to use passwords with more than 8 significant characters. Long passwords allow the use of “passphrases” such as **This is my password string**. This allows easy to remember passwords and reduces the risk of a password being comprised by a dictionary-based attack. In XENIX and earlier UNIX system implementation, this password example is treated as if the user had only entered “**This is** ” because only the first 8 characters are used for encryption.

In SCO UNIX systems, Release 3.2, the first 8 characters are still encrypted as before. Any characters following this in the clear-text password are then encrypted in groups of up to 8 characters each, using a salt³ derived from the preceding encrypted segment, and added to the end of the encrypted string.

No check is made to verify that the user exists and is not locked out of the system, because it is not possible for the user to execute this code without being logged in.

2. To translate a long SCO UNIX system V/386, Release 3.2, encrypted password string into a form usable on earlier UNIX systems, it is only necessary to truncate it to the first 13 characters. Because earlier UNIX systems ignore clear-text characters following the first eight, the encryption matches.

SCO UNIX system V/386, Release 3.2, Version 4.0 and later, provides a configurable parameter to allow the system’s encryption mechanisms to ignore clear-text characters following the first eight. This creates complete compatibility with older systems.

3. The first element of the encrypted string is a two character salt. This is followed by 11 encrypted characters using that salt. The additional strings are also 11 encrypted characters, using a salt that is taken from the first two characters of the preceding 11.

- line 118 This check is done if either the LUID requires a password or has a password assigned to it. The **password_required** and **has_password** constants are defined in lines 46 and 47.
- lines 123-125 If the user supplies the correct password, a message is sent to the Audit Trail that the user can execute this command as super user.
- line 140 An entry is made in the super-user log if it is enabled to record that the user got super-user privileges. This log is also updated when a user successfully executes the **su(C)** command to become super user. See lines 167-207 for the definition of the **sulog** function.
- lines 141-153 These lines install safe values of **PATH** and **IFS** in the environment, reset the EGID (currently "wotan") to the current RGID, and then execute the command with any arguments supplied in the command line. The EUID remains assigned to *root* because the program is installed **setuid root**.
- line 144 The **audit_no_resource(S)** routine also prints a message to **stderr**.
- lines 145-146 **setgid(S)** is a system call that is audited automatically. If an error occurs, it is handled with the **syserror** routine (defined in lines 293-310) rather than an audit record.
- lines 147-148 Similarly, **execv(S)** is also audited automatically.

Example 4-22 loge.c (Part 7/17)

```

154 void
155 failed(
156     int code,                /* exit(S) status */
157     char *who,               /* Name of attempting user, if known */
158     char *argv[]             /* Command attempted (with arguments) */
159 ) {
160     if (who != (char *)0 || (who = cuserid((char *)0)) != (char *)0)
161         solog(who, FALSE, argv);
162     exit(code);
163     /* NOTREACHED */
164 }

165 void    log_write(FILE *, char *, char *, struct tm *, int, char **);
166 FILE    *log_open(char *);

167 void
168 solog(
169     char *who,
170     int success,
171     char *argv[]
172 ) {
173     char *logf;               /* Name of su logfile */
174     char *ctty;               /* Name of console */
175     char *ttyn;               /* Name of this tty */
176     time_t now;               /* Current time (secs) */
177     struct tm at;
178     FILE *fp;
179     char *str;

180     logf = (char *)0;
181     ctty = (char *)0;
182     if (defopen(su_defaults) == 0) {
183         if ((str = defread("SULOG=")) != (char *)0)
184             logf = strdup(str);
185         if ((str = defread("CONSOLE=")) != (char *)0)
186             ctty = strdup(str);
187         defopen((char *)0);    /*close the defaults file */
188     }
189     if (logf == (char *)0 && ctty == (char *)0)
190         return;

```

- lines 167-207 These lines define the **su****log** function called in line 140. This function adds an entry to the super-user log, much as the **su**(C) command does. The */etc/default/su* file defines files in which this information should be recorded. This process does not replace auditing. Moreover, the information is not recorded as reliably as in the Audit Trail; however, it is easier to use. It also ensures that the super-user log records all accesses to super-user privileges except for *root* login sessions at the console.
- lines 180-190 These lines scan */etc/default/su* for definitions of **SU****LOG** and **CONSOLE**. If none are found, no logging is done. The code assumes that these definitions use an absolute pathname. The **su**(C) command also requires that absolute pathnames be used, so this is legitimate.
- lines 191-196 These lines determine a probable name for the terminal the user is logged in on. This name is not quite as reliable as the terminal name recorded in the Audit Trail, but it is adequate in most cases.
- lines 197-200 These lines log this attempt in the super-user logfile.
- lines 201-206 If **CONSOLE** is defined and the user is not at that location, log this attempt on that terminal.
- lines 198, 204 The **log_write** function is defined in lines 209-233.

Example 4-23 loge.c (Part 8/17)

```

191         if ((ttyn = ttyname(0)) == (char *)0)
192             if ((ttyn = ttyname(1)) == (char *)0)
193                 if ((ttyn = ttyname(2)) == (char *)0)
194                     ttyn = "/dev/tty??";

195     now = time((time_t *)0);
196     at = *localtime(&now);

197     if (logf != (char *)0 && (fp = log_open(logf)) != (FILE *)0) {
198         log_write(fp, who, ttyn, &at, success, argv);
199         (void) fclose(fp);
200     }

201     if (ctty != (char *)0 && strcmp(ctty, ttyn) != 0 &&
202         (fp = fopen(ctty, "a")) != (FILE *)0
203     ) {
204         log_write(fp, who, ttyn, &at, success, argv);
205         (void) fclose(fp);
206     }
207 }

```

Example 4-24 loge.c (Part 9/17)

```

208 void      graphic(char *, FILE *);

209 void
210 log_write(
211     FILE *fp,                /* Log file (opened to append) */
212     char *who,               /* Name of user responsible */
213     char *where,             /* Location (terminal) */
214     struct tm *when,         /* Date and time */
215     int success,             /* TRUE if command will be run */
216     char *argv[]            /* Command (and arguments) run */
217 ) {
218     char *devn;              /* Basename of terminal <where> */

219     if ((devn = strrchr(where, '/')) != (char *)0)
220         devn++;
221     else
222         devn = where;
223     (void) fprintf(fp, "LOGE %.2d/%.2d %.2d:%.2d %c %s %s-root",
224         when->tm_mon+1, when->tm_mday, when->tm_hour, when->tm_min,
225         success ? '+' : '-', devn, who
226     );
227     while (*argv != (char *)0) {
228         putc(' ', fp);
229         graphic(*argv, fp);
230         argv++;
231     }
232     putc('\n', fp);
233 }

```

lines 209-233 This **log_write** function is called in lines 198 and 204. It writes an entry to the defined log file, prefixing each entry with “LOGE” rather than “SU” to distinguish these entries from those written by the **su(C)** command. The entry also includes the command run and the arguments used.

line 229 Lines 234-255 define the **graphic** function which prints a string using all readable, printable ASCII characters.

Example 4-25 loge.c (Part 10/17)

```

234 void
235 graphic(
236     char *s,
237     FILE *fp
238 ) {
239     char c;

240     while ((c = *s++) != '\0') {
241         if ( ! isascii(c) ) {
242             (void) fputs("M-", fp);
243             c = toascii(c);
244         }
245         if (c == 0177) {
246             putc('^', fp);
247             c = '?';
248         }
249         else if (iscntrl(c)) {
250             putc('^', fp);
251             c |= 0100;
252         }
253         putc(c, fp);
254     }
255 }

```

lines 234-255 Line 229 calls the **graphic** function to output a string that uses only printable characters. It assumes that the character set in use is 7-bit ASCII only. Any character that is not a printable 7-bit ASCII character is converted to a printable character. Some utilities do not convert the tab (`\t`), bell (`\007`), and newline (`\n`) characters, but this function does to keep the log reasonably neat. If this program were going to be run on terminals that support an alternate code set, additional character conversions would be required.

Example 4-26 loge.c (Part 11/17)

```
.....
256 char      *cfs_to_str(int);
257 FILE *
258 log_open(
259     char *logfile
260 ) {
261     int dei; /* Database Error Indication*/

262     if (eaccess(logfile, F_OK)) {
263         dei = create_file_securely(logfile, AUTH_VERBOSE,
264             "record attempt to run program as superuser"
265         );
266         if (dei != CFS_GOOD_RETURN) {
267             (void) fprintf(stderr, "%s: %s: %s",
268                 command_name, cfs_to_str(dei), logfile
269             );
270             return ((FILE *)0);
271         }
272     }
273     return (fopen(logfile, "a"));
274 }

275 struct namepair cfs_str_tab[] = {
276     "Successfully and securely created file",    CFS_GOOD_RETURN,
277     "Cannot securely create new file",          CFS_CAN_NOT_OPEN_FILE,
278     "File not listed in File Control database", CFS_NO_FILE_CONTROL_ENTRY,
279     "Cannot change mode on newly created file", CFS_CAN_NOT_CHG_MODE,
280     "Cannot set owner of newly created file",   CFS_CAN_NOT_CHG_OWNER_GROUP,
281     (char *)0,                                0
282 };

283 char *
284 cfs_to_str(
285     int code /* Return value from create_file_securely() */
286 ) {
287     struct namepair *np;

288     for (np = cfs_str_tab; np->name != (char *)0; np++)
289         if (np->value == code)
290             return (np->name);
291     return ("Unknown problem with secure file creation");
292 }
```

lines 256-274 These lines open a log file.

lines 275-292 These lines convert a `create_file_securely( )` return value to a message.

Example 4-27 loge.c (Part 12/17)

```

293 void
294 syserror(
295     int code,
296     char *fmt,
297     ...
298 ) {
299     va_list args;

300     (void) fprintf(stderr, "%s: ", command_name);

301     va_start(args, fmt);
302     (void) vfprintf(stderr, fmt, args);
303     va_end(args);

304     (void) fputs(": ", stderr);
305     if (0 < code && code < sys_nerr)
306         (void) fprintf(stderr, "%s (error %d)", sys_errlist[code], code);
307     else
308         (void) fprintf(stderr, "Unknown system error %d", code);
309     putc('\n', stderr);
310 }

```

lines 293-310 The **syserror** subroutine is used rather than the standard **perror(S)** routine because this allows the use of **printf** format strings, and **perror** does not.

Example 4-28 loge.c (Part 13/17)

```

311 int
312 check_TCB_path(
313     char *path,
314     char ptype,
315     char *desc
316 ) {
317     int result;

318     if (check_DAC_path(path, ptype, desc))
319         result = -1;
320     else if ((path = find_auth_file((char *)0, OT_FILE_CNTL)) == (char *)0)
321         result = -1;
322     else {
323         result = check_DAC_path(path, TCB_FILE, "File Control database");
324         free(path);
325     }
326     return (result);
327 }

```

- lines 311-327 The **check_TCB_path** subroutine and the **check_DAC_path** routine (defined in lines 328-384) attempt to discover integrity violations. **check_TCB_path** verifies a pathname of a file listed in the File Control database, which in turn checks the integrity of the File Control database itself.
- lines 313-315 The subroutine takes three arguments, specifying, in turn, the full pathname of a file that is listed in the File Control Database (line 313), the file type (line 314; valid values are listed in lines 26-28), and a description of the file.
- line 313 The *path* name is assumed to be both canonical, meaning it does not contain consecutive backslash (\) characters, and absolute, meaning that it begins with slash (/).
- line 326 **check_TCB_path** returns 0 if the integrity check finds no problems, and non-0 otherwise. **check_TCB_path** can be fooled by pathnames containing “.”; however, this program is careful to not pass any such pathnames to this routine.

Example 4-29 loge.c (Part 14/17)

```

328 int
329 check_DAC_path(
330     char *path,
331     char ptype,
332     char *desc
333 ) {
334     char c, *temp, *cp, *text, buf[ARBSTRSZ], type;
335     struct pr_file *prf;
336     enum DAC_status sts;
337
338     if ((temp = strdup(path)) == (char *)0) {
339         audit_no_resource(path, OT_MEMORY, "cannot check pathname");
340         return (-1);
341     }
342
343     cp = temp + 1;          /* Assume absolute path (*temp is "/") */
344     (void) sprintf(buf, "Path leading to %s", desc);
345     text = buf;
346     type = TCB_DIRECTORY;
347     for (;;) {
348         if (cp == (char *)0 || (c = *cp) == ' ') {
349             text = desc;
350             type = ptype;
351         }
352         else {
353             c = *cp;
354             *cp = ' ';
355         }
356         if ((prf = getprfinam(temp)) == (struct pr_file *)0)
357             sts = NotListed;
358         else if (type && prf->uflg.fg_type && *prf->uflg.fd_type != type)
359             sts = WrongDAC;
360         else
361             sts = compare_DAC_attributes(temp, prf);

```

- lines 328-384 The **check_DAC_path** subroutine checks the attributes of an entire path; it checks the components of a path to see that the DAC is correct. In combination with the **check_TCB_path** routine (defined in lines 311-327), this routine verifies the integrity of the File Control database itself.
- lines 337-340 These lines make a working copy of the pathname that can be modified.
- lines 341-384 This code “walks” down the working pathname, verifying that each intermediate directory is indeed a directory, until the entire path is checked against the File Control database.

Example 4-30 loge.c (Part 15/17)

```
360         switch (sts) {
361             case MatchesDAC:
362             case ExceedsDAC:
363                 break;
364             case NotListed:
365                 sa_audit_security_failure(
366                     OT_FILE_CNTL,
367                     (long)NotListed, (long)MatchesDAC,
368                     temp, text
369                 );
370                 free(temp);
371                 return (-1);
372             case WrongDAC:
373             default:
374                 audit_lax_file(temp, text);
375                 free(temp);
376                 return (-1);
377         }
378         if (cp == (char *)0 || (*cp = c) == '\0')
379             break;
380         cp = strchr(cp + 1, '/');
381     }
382     free(temp);

383     return (0);
384 }
```

Example 4-31 loge.c (Part 16/17)

```

385 enum DAC_status
386 compare_DAC_attributes(
387     char *path,
388     register struct pr_file *prf
389 ) {
390     struct stat stbuf;
391     mode_t f_mode, d_mode;
392     char f_type;

393     if (prf == (struct pr_file *)0)
394         return (NotListed);

395     if (stat(path, &stbuf))
396         return (NoSuchFile);

397     if (prf->uflg.fg_uid && prf->ufld.fd_uid != stbuf.st_uid)
398         return (WrongDAC);
399     if (prf->uflg.fg_gid && prf->ufld.fd_gid != stbuf.st_gid)
400         return (WrongDAC);

401     if (prf->uflg.fg_type) {
402         switch (stbuf.st_mode & S_IFMT) {
403             case S_IFDIR:  f_type = 'd';    break;
404             case S_IFREG:  f_type = 'r';    break;
405             case S_IFCHR:  f_type = 'c';    break;
406             case S_IFBLK:  f_type = 'b';    break;
407 #ifdef S_IFIFO
408             case S_IFIFO:  f_type = 'f';    break;
409 #endif
410 #ifdef S_IFSOCK
411             case S_IFSOCK: f_type = 's';    break;
412 #endif

```

- lines 385-431 The **compare_DAC_attributes** subroutine compares the File Control database to the actual disk file. The return value reports the result:
- | | |
|-------------------|---|
| ExceedsDAC | Existing file grants less access than required. |
| NoSuchFile | Disk file does not exist. |
| NotListed | File is not listed in File Control database. |
| MatchesDAC | Existing file matches the required DAC exactly. |
| WrongDAC | Existing file does not match the required DAC. |
- As a general rule, everything listed in the File Control database is part of the TCB.
- lines 397-400 If the user or group of the disk file is not listed in the database, it is treated as a “wildcard” feature that is matched by all users or groups.
- lines 401-420 If the database specifies a file type (regular, directory, and so on), compare it to the file type defined in the disk file’s inode.

Example 4-32 loge.c (Part 17/17)

```

413 #ifdef S_IFLNK
414             case S_IFLNK: f_type = 'l';      break;
415 #endif
416             default:      return (WrongDAC);
417         }
418         if (*prf->ufl.d.fd_type != f_type)
419             return (WrongDAC);
420     }

421     f_mode = (stbuf.st_mode & ~S_IFMT);
422     if (prf->ufl.fg_mode)
423         d_mode = (prf->ufl.d.fd_mode & ~S_IFMT);
424     else
425         d_mode = 0;
426     if (d_mode == f_mode)
427         return (MatchesDAC);
428     else if ((~d_mode & f_mode) == 0)
429         return (ExceedsDAC);

430     return (WrongDAC);
431 }

```

lines 421-431 If the database lists an access mode for the file, compare it to the actual mode of the file. If the database does not list an access mode for the file, this routine assumes 000 permissions (no access at all). Note that the modes do not have to compare exactly; such things as a sticky bit or lock set against the file may modify the disk file's mode but not be reflected in the File Control database.

Chapter 5

Using the event manager API

The event manager API is a collection of utilities, system calls, and header files that are used to develop applications that use mice and other graphics input devices. The event manager API is completely separate from the control of the mouse provided by the X server, but applications that use the event manager API can be run in an SCO Graphical Environment without conflicting with the mouse control provided by the X server.

Data from graphics input devices like mice is received by the application in the form of “events”. An event represents a movement or action on the part of the graphics input device. For example, moving a mouse generates a series of signals from the mouse that are interpreted as discrete events. An “up” motion of the mouse is interpreted as an “up” event. Events are accumulated in, and read from, an “event queue”. Mouse support is provided by a series of utilities and system services collectively known as the event manager. The event manager consists of kernel functionality to read and manage graphic input device data, an event driver, and an Application Programming Interface.

The event manager API allows applications software to be portable to systems with different event devices, such as a different brand of mouse or a mouse attached to a different port. The interface provides routines for opening and accessing an event queue. A variety of graphics input devices, such as mice, bitpads or the keyboard, may be used with the event driver. Note, however, that the event manager API is not part of any current industry standards so may not be portable to other UNIX platforms.

Speed is of top priority in graphics applications, so the event driver and the system routines are optimized for speed of access. Applications written with the event manager API should also be optimized for speed.

Applications which use the Event Manager API, and which run under **SCOterm** windows, will not read events from the event queue unless **SCOterm** allows it. This feature can be toggled on and off by pressing (Ctrl)(Alt)(Bksp); when off, mouse movements, button clicks and other events will be interpreted by **SCOterm** and not passed on to the application. When active, the reverse occurs; events are placed in the event queue and are not interpreted by **SCOterm**. Furthermore the pointer will disappear and focus will remain in the **SCOterm** window, until the feature is toggled off.

Operation and semantics overview

To open an event queue, a process makes a library call. The process must indicate the classes (types) of devices it wants to use. Common classes for devices include “relative locator” devices (mice) or “absolute locator” devices (bitpads). The event manager interface then checks the configuration files to determine what physical devices are allowed to give input to the process’s terminal. The devices listed in the configuration file are “associated” with that terminal. Associated devices of the proper class are opened for events with the initial call from the process. When a device is opened for events, data from the device is translated to events, and entered into the process’s event queue.

An event device is associated with some terminals or console multi-screens. When using an event device with the console, it should be associated with each multiscreen. In order for an application to use event devices, the devices must be associated with the terminal that invoked the application. It is possible for programs that are not running on any terminal to use the mouse, but they cannot use the event manager API.

An event device is said to be “attached” to an event queue if the device has been opened and input from the device is being received by the event queue. Note that an application can choose whether or not to attach any associated event device to its event queue while it is running. For example, if both a bitpad and a mouse are associated with a terminal, it is possible for the application to open only the mouse for input, and then close the mouse and open the bitpad without exiting.

Event generating hardware

Event devices can be categorized as relative locator, absolute locator, or string. The **event(FP)** manual page lists the defined constants that represent these classes. Most mice and trackballs are examples of relative locator devices. They are termed “relative” devices because the mouse generates events when it is moved relative to its previous position. The motion of the mouse generates the events, not the final position of the mouse. Relative locations are reported as signed 32-bit quantities. Bit-pads and light pens are absolute locator devices. The pens are used on a sensitive pad, and the events are generated by the absolute position of the pen on the pad, not the relative motion of the pen. Absolute locations are reported as unsigned 32-bit quantities. The keyboard is a string graphics input device. This means that the events will be generated by pressing a key on the keyboard, such as an arrow key. Some devices can operate in multiple modes. For example, some mice can function in bitpad mode with the appropriate bitpad software.

Many event devices have buttons; others do not. Three-button devices are viewed as having a left button, a middle button, and a right button. Two-button devices do not have a middle button. One-button devices have only a middle button. Bits representing buttons are set to 1 when the button is depressed. Up to eight buttons can be read at a time. The **event(FP)** manual page lists macros used to identify buttons.

Using the event queue

The event manager API allows an application to obtain sequential input from any number of event devices. Devices like mice, trackballs, bitpads, and the keyboard can be assigned to one or more terminals or multiscreens. When a process opens an event queue with more than one device attached, input from all the devices is intermixed in the event queue. This allows a process to “sleep on multiple devices” without recourse to the **select(S)** system call.

An application makes a call to a library routine to open an event queue. That call attaches the event devices to the event queue. In general, an application may be using multiple devices. For example, an application would typically want to use a mouse and the keyboard. In cases where there are several devices attached to the terminal, for example multiple mice, calls are provided for listing the attached devices and including them or excluding them on an individual basis.

The event driver maintains a circular queue of events. “Circular” means that if the total number of events in the queue exceeds the maximum number of events storable, that the oldest event is removed from the queue and an error condition is returned.

The event queue is “fed” by all the devices attached to the queue. An input mask may also be set which prevents certain classes of events from being inserted into the queue. For example, you may wish to exclude button events or movement events during some portions of your program.

The application program reads the event queue by getting a pointer to the next event. When the application has copied or finished examining the event, it pops the event queue using the **ev_pop(S)** routine. The next **ev_read(S)** call will return a pointer to the next event in the queue, or **NULL** if the queue is empty. Multiple reads that are not separated by a call to pop the queue return the same pointer.

Event manager calls

Table 5-1, “Event manager library calls” (this page) lists the event manager routines for managing and accessing an event queue and the attached devices.

Table 5-1 Event manager library calls

Function	Description
ev_init	initialize event queue
ev_open	open the event queue, attach event devices
ev_close	close the event queue and all attached devices
ev_count	return the number of events in the queue
ev_read	get a pointer to the “top” event
ev_pop	pop the “top” event off the queue
ev_block	sleep until the queue is nonempty
ev_flush	pop (discard) all events in the queue
ev_getdev	get a list of devices feeding the queue
ev_gindev	exclude or later re-include a event device
ev_setemask	mask out certain kinds of events
ev_getemask	get the current event mask
ev_suspend	suspend the active event queue (make it inactive)
ev_resume	resume the suspended event queue

All of the event manager routines that return an integer return a negative number if they fail. All of these routines with the exception of **ev_init(S)** and **ev_open(S)** fail if the calling program did not first initialize an event queue with the **ev_init(S)** routine.

The constants and types used in the event manager are defined in */usr/include/mouse.h*.

A sample program

The following sample program opens an event queue with a mouse (relative device) and the keyboard (string device). It prints out information about events as they enter the queue. Note that depending on your system, you may have to configure system event parameters.

This program illustrates basic event manager functionality. It does not use the routines for querying or modifying the devices or event masks for the queue, nor the routines for suspending the event queue.

```

1 #include <stdio.h>
2 #include <signal.h>
3 #include <sys/machdep.h>
4 #include <sys/types.h>
5 #include <sys/param.h>
6 #include <sys/sysmacros.h>
7 #include <sys/page.h>
8 #include <sys/event.h>
9 #include <mouse.h>

10 #define DEL      0x7f

11 char    progame[80];
12 extern int errno;

13 main(argc,argv)
14 int  argc;
15 char *argv[];
16 {
17     int      qfd,finish(),report();
18     EVENT    *evp;
19     int      ret;
20     dmask_t dmask = D_STRING | D_REL | D_BUTTON;
21     extern int ev_errlev;

```

- line 3 The `<sys/machdep.h>` header file includes definitions for the event driver `IOCTL`'s and the mouse discipline.
- lines 4-6 These header files are required in virtually every device driver to define data types, structures, and macros used to access devices directly. Because the event manager API is so closely tied to the driver, these header files are also required in any program that uses the event manager.
- lines 8-9 These two header files must be `#included` in this order in any program that uses the event manager.
- line 10 This maps the `DEL` variable to the hardware register that is normally associated with the `<Del>` key. Line discipline zero functionality such as signals is not available to the keyboard, because it is running under the event line discipline. Consequently, the `<Del>` character loses its special meaning. To solve this problem, the program tests each character against `DEL`. This variable will be used in line 66.
- line 20 The device mask is used as an argument to the `ev_open(S)` call in line 32. It is a bitwise `OR`'ing of any values that describe the category of devices that are to be opened. This device mask specifies relative locator devices (such as mice); absolute locator devices (such as bitpads); and devices that generate character strings. A fourth value, `D_OTHER`, can be used to specify event devices that are not within the other categories. These values are defined on the `event(FP)` manual page.

```

32 ev_errlev = 1;
33 strcpy(progname,argv[0]);
34 signal(SIGINT,finish);
35 signal(SIGSEGV, report);
36 signal(SIGSYS, report);

37 printf("\n");
38 ret = ev_init();
39 printf("init == %d\n", ret);
40 if ( ret < 0 )
41     finish();
42 qfd = ev_open(&dmask);
43 printf("open == %d\n", qfd);
44 if ( qfd < 0 )
45     fail("could not open event queue");
46 if ( dmask != (D_STRING | D_REL | D_BUTTON))
47     fail("could not attach mouse and keyboard");
48     while ( !(ev_block()) )
49         if ( (evp = ev_read()) != (EVENT *) NULL ) {
50             evprint(evp);
51             ev_pop();
52         }
53     return 0;
54 }
55

56 fail(s)
57 char *s;
58 {
59     extern int errno;

60     printf("s(%d): %s\n",progname,errno,s);
61     finish();
62     return 0;
63 }

```

line 22 Turn on the event manager diagnostics. **ev_errlev** is an error-level variable that can be set to a nonzero value to have the event manager print out more verbose diagnostics that can be used for debugging. By default, **ev_errlev** is set to 0. Once the code is thoroughly debugged, this line should be removed to improve the performance of the program.

lines 28-31 **ev_init(S)** reads the system event configuration files and initializes the event queue. It must be the first event manager routine called in the program. If there is a syntax error or inconsistency in the configuration files, it returns a small negative integer; if it successfully initializes the event queue, it returns 1.

lines 32-37 **ev_open(S)** opens an event queue for the process. All the event devices that are associated with the terminal and whose class is masked in are opened and added to the queue.

The argument to the **ev_open** routine points to the device mask that was defined in line 20. The function tries to attach devices of the indicated type(s) to the vent queue. If **ev_open** cannot find any devices to attach to the queue, it closes the queue and returns a negative number indicating an error. If it is able to attach devices, it sets the mask to indicate what kinds of devices it has found. So line 36 verifies that the mask is still set as it was in line 20; if it is not, the program exits and returns an error message using the **fail()** function defined in lines 47-54.

If **ev_open** is successful, it returns a file descriptor that is used with the **select(S)** system call. It should not be used for reading or writing.

lines 38-45 At this point, events should be enqueued. This section of code illustrates the standard practice of using **ev_block(S)** to block until there is an event in the queue, then read the event with **ev_read(S)**, and then remove the last read event from the queue with **ev_pop(S)**.

ev_read takes no arguments. It reads the next element in the event queue and returns a pointer to the next element in the vent queue, or to **NULL** if the queue is empty. Multiple calls to this routine return the same pointer until a call to **ev_pop** is made. **ev_pop** pops the next element off the queue. As soon as this routine is called, the pointer returned by the most recent call to **ev_read** must be considered invalid, because the event driver can overwrite the pointed area with new information.

ev_pop fails if the queue is empty. Otherwise, it returns a number that indicates how many events have been lost to queue overrun since the last **ev_pop**. The queue is of a fixed size (determined by the **EVDEVSPERQ** tunable parameter). If the application does not read the queue fast enough, events may be lost; this is called “queue overrun” because the event driver would overrun the tail of the queue if it wrote new events. The event driver provides a counter that indicates the number of events it has lost. That counter is cleared when it succeeds in writing an event. An **ev_pop** call always causes a new slot to be available, which clears the overrun counter. **ev_pop** returns the old overrun counter so that an application is aware if events are being lost.

lines 46-53 The **fails()** function is called in line 35 to indicate an error condition.

```
54 evprint(ev)
55 EVENT    *evp;
56 {
57     static long time=-1;

58     if (time == -1)
59         time = evp->timestamp;
60     if (EV_TAG(*evp) & T_STRING)    {
61         printf("time(%d) tag(%d) bufsize(%d) buf(%x)\n",
62             EV_TIME(*evp)-time,
63             EV_TAG(*evp),
64             EV_BUFCNT(*evp),
65             EV_BUF(*evp)[0]);
66         if (EV_BUF(*evp)[0] == DEL)
67             finish();
68     }
69     else    {
70         printf("time(%ld) tag(%d) butts(%d) x(%ld) y(%ld)\n",
71             EV_TIME(*evp) - time,
72             EV_TAG(*evp),
73             EV_BUTTONS(*evp),
74             EV_DX(*evp),
75             EV_DY(*evp));
76         if (EV_BUTTONS(*evp) == 7)
77             finish();
78     }
79     return 0;
80 }

81 finish()
82 {
83     ev_close();
84     printf("done\n");
85     exit(0);
86 }

87 report(s)
88 int s;
89 {
90     printf("got signal %d\n", s);
91     finish();
92 }
```

- lines 54-78 This section of code illustrates how a program can access information about a specific event by using the event extracting constants that are listed on the **event(FP)** manual page.
- lines 81-86 This subroutine closes the event queue and exits the program. It is called in line 24 when a **SIGINT** signal is received, in line 51 as part of the **fail()** routine, in line 91 as part of the **report()** routine, and at other points in the code where the program needs to exit without returning an error message.

Chapter 6

Job control under ksh

The UNIX operating system Korn shell utility, **ksh(C)** supports job control for applications that execute under this shell. ¹ Job control works for almost all programs; however, screen-oriented programs using **curses(S)** must make special provisions to utilize job control. The example code on the following pages illustrates how to write a **curses** application that uses job controls to switch contexts between **curses** mode and a shell line mode.

For more background about using Korn Shell job control, see David Korn's *Korn Shell* book. To understand this sample code, you should also understand the signal mechanism.

To compile this program, use the following command:

```
cc file.c -lcurses
```

This example assumes that you have the source code for the program to be run under job control and can rewrite it as shown here. If that is not the case, you can implement job control by writing a program similar to the one shown here and inserting your program sections as shown in the following table:

Program section	Insert code after line #
Initialization information	22
Main body of program	23 (replacing the while loop)
Code to set terminal to program's screen mode	37
Code to set terminal to the new context	41

1. Note that job control is not supported under the Korn shell on XENIX releases.

```

1 #include <signal.h>
2 #include <curses.h>

3 extern int SigHandler( int );
4 struct sigaction signal_action, old_action;

5 main()
6 {
7     sigset_t set, old_set;

8     signal_action.sa_handler = SigHandler;
9     signal_action.sa_flags = 0;

10    sigemptyset( &set);
11    signal_action.sa_mask = set;

```

lines 1 and 2 Header files are included for the program. *signal.h* provides the signal values, such as **SIGTSTP** shown in line 12. *curses.h* provides support for the **curses** routines and macros used in this program.

line 4 Declare an instance of the **sigaction** structure. The structure is as follows:

```

struct sigaction {
    void    (*sa_handler)();    /* signal handler */
    sigset_t sa_mask;           /* signal mask   */
    int     sa_flags;           /* signal flags  */
};

```

line 7 Declare the **set** and **old_set** variables to **sigset_t**, which is defined as a **long** in the */usr/include/signal.h* header file. **set** is a set of signals.

line 8 Assign the name of the signal handler subroutine to the **sa_handler** member of the **sigaction** structure.

line 9 Turn off the flags field. This field is only used for the **SIGCHLD** signal, which is not used in this example.

line 10 Create an empty set of signals as the first step in blocking those signals associated with a terminal device. It is necessary to block these signals so that the signal handler routine can be established and so that **curses** can be initialized before signals are received.

line 11 Set the **signal_action.sa_mask** to be empty.

```

12      sigaddset( &set, SIGTSTP );
13      sigaddset( &set, SIGTTIN );
14      sigaddset( &set, SIGCONT );
15      sigaddset( &set, SIGTTOU );
16      sigprocmask( SIG_SETMASK, &set, &old_set );

17      sigaction( SIGCONT, &signal_action, NULL );
18      signal_action.sa_mask = set;

```

lines 12 to 16 Add the signals to be blocked. When signals are blocked they are postponed, but not ignored. The call to **sigprocmask** in line 16 actually causes the signals in the set to be blocked. The signals are blocked so that program initialization can occur prior to actually receiving a signal. **sigprocmask** also saves the old mask value into **old_set**.

The **sigaddset** calls assign values to the **set** mask. The signals are:

- **SIGTSTP** occurs when an application sends **<Ctrl> z** to push the current foreground task into the background. The terminal is requesting that the current program stop.
- **SIGTTIN** occurs when a background process attempts to read from the terminal.
- **SIGCONT** continues a stopped process.
- **SIGTTOU** occurs when a background process attempts to write to the terminal, if the **TOSTOP** mode is set in the **c_lflags** member of the **termios** structure. **SIGTTOU** can also occur if a background process attempts to change the terminal **termios** members with an **ioctl**.

line 17 Now that the signals from the terminal device are blocked, the signal handler routine can be initialized. The **sigaction** call sets a function pointer in the kernel for the signal handler routine (**SigHandler**) and permits the signal handler routine to be executed only when the specified signal, in this case, **SIGCONT** is received. The main difference between **signal** and **sigaction** is that **sigaction** utilizes the information in the **sigaction** structure. When the signal handler is called, none of the signals specified in the mask are received.

line 18 Set the **sigaction** structure mask to the values held in the signal **set**. This is a necessary initialization step prior to calling **sigaction** to establish the conditions under which the signal handler is called for the next set of signals.

```
19      sigaction( SIGTIN, &signal_action, &old_action );
20      sigaction( SIGTTOU, &signal_action, NULL );
21      sigaction( SIGTSTP, &signal_action, NULL );

22      initscr();
23      sigprocmask( SIG_SETMASK, &old_set, NULL );

24      while (pause())
25      {
26          printf( "Got a Signal\n" );
27          refresh();
28      }
29 }
```

- lines 19 to 21 Execute **sigaction** for any possible signals from the terminal device. This ensures that the signal handler executes for each signal value.
- line 22 Now that the signal handler is established, the **curses** application can be started. Initialize the **curses** data structures and get terminal information for use by the **curses** routines that follow.
- line 23 With the **curses** application started, use **sigprocmask** to let the signals be received.
- lines 24 to 28 Wait until a signal is received in line 24 using **pause**. When a signal arrives, display the Got a Signal message and rewrite the **curses** window.

```

30 SigHandler( sig )
31 int sig;
32 {
33     if (sig == SIGCONT)
34     {
35         sigaction( SIGTTIN, &signal_action, NULL );
36         sigaction( SIGTTOU, &signal_action, NULL );
37         sigaction( SIGTSTP, &signal_action, NULL );
38         reset_prog_mode();
39         printf("Returned to curses line modes.\n");
40     } else {
41         sigaction( sig, &old_action, NULL );
42         printf( "Returning to shell line modes. \n");
43         refresh();
44         reset_shell_mode();
45         kill( getpid(), sig );
46     }
47 }

```

- line 30 The **SigHandler** signal handler routine operates in two conditions. Either the **SIGCONT** signal occurs and process continues in **curses** mode, or **SIGTSTP**, **SIGTTIN**, or **SIGTTOU** occurs and the program performs a context shift to a different mode, in this case, the shell line mode.
- lines 33 to 39 The **SIGCONT** signal occurred and the **curses** process continues. The signal handler is now instructed to be executed when the three other signals occur. The terminal device is returned to **curses** mode and a message displayed in the window.
- lines 41 to 45 One of the **SIGTSTP**, **SIGTTIN**, or **SIGTTOU** signals occurred. In line 41, **sigaction** sets the conditions for the signal handler back to the previously stored value. A message displays in the window indicating that the mode is now changing. The mode is then changed in line 44, and finally the signal is sent to the process.

The following manual pages provide additional information about the routines and structures described in the example:

Routine	Manual Page	Description
getpid	getpid(S)	get process group ID
initscr	curses(S)	initialize curses terminal and structures
kill	kill(S)	sends a signal to the process group
pause	pause(S)	suspend process until signal is received
printw	curses(S)	display message in curses window
refresh	curses(S)	restores screen display
reset_prog_mode	curses(S)	restore to curses mode
reset_shell_mode	curses(S)	restore to non- curses mode
sigaction	sigaction(S)	examine and change signal action
sigaddset	sigset(S)	add signal to signal mask
sigemptyset	sigset(S)	get empty signal set and clear mask
signal	signal(S)	traditional signal handler
sigprocmask	sigprocmask(S)	block signals from mask
termio	termio(M)	terminal interface
termios	termios(M)	POSIX terminal interface

Chapter 7

Common Object File Format (COFF)

This chapter describes the Common Object File Format (COFF). COFF is the format of the output file produced by the assembler, **as**(CP), and the link editor, **ld**(CP).

The following are some key features of COFF:

- applications can add system-dependent information to the object file without causing access utilities to become obsolete
- space is provided for symbolic information used by debuggers and other applications
- programmers can modify the way the object file is constructed by providing directives at compile time

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file contains:

- a file header
- optional header information
- a table of section headers
- data corresponding to the section headers
- relocation information
- line numbers
- a symbol table
- a string table

Table 7-1, “Object file format” (page 132) shows the overall structure.

Table 7-1 Object file format

FILE HEADER
Optional Information
Section 1 Header
...
Section <i>n</i> Header
Raw Data for Section 1
...
Raw Data for Section <i>n</i>
Relocation Info for Sect. 1
...
Relocation Info for Sect. <i>n</i>
Line Numbers for Sect. 1
...
Line Numbers for Sect. <i>n</i>
SYMBOL TABLE
STRING TABLE

The last four sections (relocation, line numbers, symbol table, and the string table) may be missing if the program is linked with the **-s** option of the **ld** command, or if the line number information, symbol table, and string table are removed by the **strip(CP)** command. The line number information does not appear unless the program is compiled with **cc -g**, in the *Programmer's Reference Manual Volume 1*. Also, if there are no unresolved external references after linking, the relocation information is no longer needed and is absent. The string table is also absent if the source file does not contain any symbols with names longer than eight characters.

An object file that contains no errors or unresolved references is considered executable.

Definitions and conventions

Before proceeding further, you should become familiar with the following terms and conventions.

Sections

A section is the smallest portion of an object file that is relocated and treated as one separate and distinct entity. In the most common case, there are three sections named `.text`, `.data`, and `.bss`. Additional sections accommodate comments, multiple text or data segments, shared data segments, or user-specified sections. However, the operating system loads only `.text`, `.data`, and `.bss` into memory when the file is executed.

NOTE It is a mistake to assume that every COFF file will have a certain number of sections, or to 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 it from file and section headers in the file.

Physical and virtual addresses

The physical address of a section or symbol is the offset of that section or symbol from address zero of the address space. The term physical address as used in COFF does not correspond to general usage. The physical address of an object is not necessarily the address at which the object is placed when the process is executed. For example, on a system with paging, the address is located with respect to address zero of virtual memory and the system performs another address translation. The section header contains two address fields, a physical address, and a virtual address; these addresses are equivalent on UNIX systems,

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 exact same computer on which the object file is being created.

File header

The file header contains the 20 bytes of information shown in Table 7-2, “File header contents” (this page). The last 2 bytes are flags that are used by `ld` and object file utilities.

Table 7-2 File header contents

Bytes	Declaration	Name	Description
0-1	unsigned short	<code>f_magic</code>	Magic number
2-3	unsigned short	<code>f_nscns</code>	Number of sections
4-7	long int	<code>f_timdat</code>	Time and date stamp indicating when the file was created, expressed as the number of elapsed seconds since 00:00:00 GMT, January 1, 1970
8-11	long int	<code>f_symptr</code>	File pointer containing the starting address of the symbol table
12-15	long int	<code>f_nsyms</code>	Number of entries in the symbol table
16-17	unsigned short	<code>f_opthdr</code>	Number of bytes in the optional header
18-19	unsigned short	<code>f_flags</code>	Flags (see Table 7-3, “File header flags” (page 135))

Magic numbers

The magic number specifies the target machine on which the object file is executable.

Flags

The last 2 bytes of the file header are flags that describe the type of the object file. Currently defined flags are found in the header file `filehdr.h` and are shown in Table 7-3, “File header flags” (page 135).

Table 7-3 File header flags

Mnemonic	Flag	Meaning
F_RELFLG	00001	Relocation information stripped from the file
F_EXEC	00002	File is executable (i.e., no unresolved external references)
F_LNNO	00004	Line numbers stripped from the file
F_LSYMS	00010	Local symbols stripped from the file
F_AR16WR	0000200	16-bit byte reversed word
F_AR32WR	0000400	32-bit byte reversed word

File header declaration

The C structure declaration for the file header is shown below. This declaration may be found in the header file *filehdr.h*.

```

struct filehdr
{
    unsigned short    f_magic; /* magic number */
    unsigned short    f_nscns; /* number of section */

    long              f_timdat; /* time and date stamp */

    long              f_symptr; /* file ptr to symbol table */

    long              f_nsyms; /* number entries in the symbol table */

    unsigned short    f_opthdr; /* size of optional header */

    unsigned short    f_flags; /* flags */
};

#define FILHDR struct filehdr
#define FILHSZ sizeof(FILHDR)

```

Optional header information

The template for optional information varies among different systems that use COFF. Applications place all system-dependent information into this record. This allows different operating systems access to information that only that operating system uses without forcing all COFF files to save space for that information. General utility programs (for example, the symbol table access library functions, and the disassembler) are made to work properly on any common object file. This is done by seeking past this record using the size of optional header information in the file header field `f_opthdr`.

Standard UNIX system *a.out* header

By default, files produced by the link editor for a UNIX system always have a standard UNIX system *a.out* header in the optional header field. The UNIX system *a.out* header is 28 bytes. The fields of the optional header are described in Table 7-4, “Optional header contents” (this page).

Table 7-4 Optional header contents

Bytes	Declaration	Name	Description
0-1	short	magic	Magic number
2-3	short	vstamp	Version stamp
4-7	long int	tsize	Size of text in bytes
8-11	long int	dsize	Size of initialized data in bytes
12-15	long int	bsize	Size of uninitialized data in bytes
16-19	long int	entry	Entry point
20-23	long int	text_start	Base address of text
24-27	long int	data_start	Base address of data

Whereas the magic number in the file header specifies the machine on which the object file runs, the magic number in the optional header supplies information telling the operating system on that machine how that file should be executed. The magic numbers recognized by SCO operating systems are given in Table 7-5, “UNIX system magic numbers” (page 137).

Table 7-5 UNIX system magic numbers

Value	Meaning
0407	Text segment is not write-protected or sharable; data segment is contiguous with the text segment.
0410	Data segment starts at the next segment following the text segment and the text segment is write-protected.
0413	Text and data segments are aligned within <i>a.out</i> so it can be directly paged.
0443	Defines <i>a.out</i> to be a target shared library.

Optional header declaration

The C language structure declaration currently used for the UNIX system *a.out* file header is shown below. This declaration may be found in the header file *aouthdr.h*.

```
typedef struct aouthdr
{
    short    magic;        /* magic number */
    short    vstamp;       /* version stamp */
    long     tsize;        /* text size in bytes, padded */
                          /* to full word boundary */

    long     dsize;        /* initialized data size */

    long     bsize;        /* uninitialized data size */

    long     entry;        /* entry point */

    long     text_start;    /* base of text for this file */

    long     data_start    /* base of data for this file */
} AOUTHDR;
```

Section headers

Every object file has a table of section headers to specify the layout of data within the file. The section header table consists of one entry for every section in the file. The information in the section header is described in Table 7-6, “Section header contents” (this page).

Table 7-6 Section header contents

Bytes	Declaration	Name	Description
0-7	char	s_name	8-character null padded section name
8-11	long int	s_paddr	Physical address of section
12-15	long int	s_vaddr	Virtual address of section
16-19	long int	s_size	Section size in bytes
20-23	long int	s_scnptr	File pointer to raw data
24-27	long int	s_relptr	File pointer to relocation entries
28-31	long int	s_lnnoptr	File pointer to line number entries
32-33	unsigned short	s_nreloc	Number of relocation entries
34-35	unsigned short	s_nlnno	Number of line number entries
36-39	long int	s_flags	Flags (see Table 7-7, “Section header flags” (page 139))

The size of a section is padded to a multiple of 4 bytes. File pointers are byte offsets that can be used to locate the start of data, relocation, or line number entries for the section. They can be readily used with the function **fseek(S)**.

Flags

The lower 2 bytes of the flag field indicate a section type. The flags are described in Table 7-7, “Section header flags” (this page).

Table 7-7 Section header flags

Mnemonic	Flag	Meaning
STYP_REG	0x00	Regular section (allocated, relocated, loaded)
STYP_DSECT	0x01	Dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	0x02	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0x04	Grouped section (formed from input sections)
STYP_PAD	0x08	Padding section (not allocated, not relocated, loaded)
STYP_COPY	0x10	Copy section (for a decision function used in updating fields; not allocated, not relocated, loaded, relocation and line number entries processed normally)
STYP_TEXT	0x20	Section contains executable text
STYP_DATA	0x40	Section contains initialized data
STYP_BSS	0x80	Section contains only uninitialized data
STYP_INFO	0x200	Comment section (not allocated, not relocated, not loaded)
STYP_OVER	0x400	Overlay section (relocated, not allocated, not loaded)
STYP_LIB	0x800	For <code>.lib</code> section (treated like STYP_INFO)

Section header declaration

The C structure declaration for the section headers is shown below. This declaration may be found in the header file *scnhdr.h*.

```
struct scnhdr
{
    char        s_name[8];        /* section name */
    long        s_paddr;          /* physical address */
    long        s_vaddr;          /* virtual address */
    long        s_size;           /* section size */
    long        s_scnptr;         /* file ptr to section raw data */

    long        s_relptr;         /* file ptr to relocation */

    long        s_lnnoptr;        /* file ptr to line number */

    unsigned short s_nreloc;      /* number of relocation entries */

    unsigned short s_nlnno;      /* number of line number entries */

    long        s_flags;          /* flags */
};

#define SCNHDR struct scnhdr
#define SCNHSZ sizeof(SCNHDR)
```

.bss section header

The one deviation from the normal rule in the section header table is the entry for uninitialized data in a **.bss** section. A **.bss** section has a size and symbols that refer to it, and symbols that are defined in it. At the same time, a **.bss** section has no relocation entries, no line number entries, and no data. Therefore, a **.bss** section has an entry in the section header table but occupies no space elsewhere in the file. In this case, the number of relocation and line number entries, as well as all file pointers in a **.bss** section header, are 0. The same is true of the STYP_NOLOAD and STYP_DSECT sections.

Sections

Table 7-1, “Object file format” (page 132) shows that section headers are followed by the appropriate number of bytes of text or data. The raw data for each section begins on a 4-byte boundary in the file.

Link editor `SECTIONS` directives allow users to do the following, among other things:

- describe how input sections are to be combined
- direct the placement of output sections
- rename output sections.

If no `SECTIONS` directives are given, each input section appears in an output section of the same name. For example, if a number of object files, each with a `.text` section, are linked together, the output object file contains a single `.text` section made up of the combined input `.text` sections.

Relocation information

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the format described in Table 7-8, “Relocation section contents” (this page).

Table 7-8 Relocation section contents

Bytes	Declaration	Name	Description
0-3	long int	<code>r_vaddr</code>	(Virtual) address of reference
4-7	long int	<code>r_symndx</code>	Symbol table index
8-9	unsigned short	<code>r_type</code>	Relocation type

The first 4 bytes of the entry are the virtual address of the text or data to which this entry applies. The next field is the index, counted from 0, of the symbol table entry that is being referenced. The type field indicates the type of relocation to be applied.

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated. The currently recognized relocation types are given in Table 7-9, “Relocation types” (this page).

Table 7-9 Relocation types

Mnemonic	Flag	Meaning
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_DIR16 *	01	Direct, 16-bit reference to a symbol’s virtual address.
R_REL16 *	02	“PC-relative”, 16-bit reference to a symbol’s virtual address. Relative references occur in instructions such as jumps and calls.
R_DIR32	06	Direct 32-bit reference to the symbol’s virtual address.
R_SEG12 *	011	Direct, 16-bit reference to the segment-selector bits of a 32-bit virtual address.
R_PCRLONG †	024	“PC-relative”, 32-bit reference to a symbol’s virtual address.

* 80286 Computer only.

† 80386 Computer only.

Relocation entry declaration

The structure declaration for relocation entries is shown below. This declaration may be found in the header file *reloc.h*.

```

struct reloc
{
    long      r_vaddr; /* virtual address of reference */
    long      r_symndx; /* index into symbol table */
    unsigned short r_type; /* relocation type */
};

#define RELOC      struct reloc
#define RELSZ      10

```

Line numbers

When invoked with `cc -g` in the *Programmer's Reference Manual Volume 1* the compiler causes an entry in the object file for every source line where a break-point can be inserted. You can then reference line numbers when using a software debugger like `sdb`(CP). All line numbers in a section are grouped by function as shown in Table 7-10, "Line number grouping" (this page).

Table 7-10 Line number grouping

symbol index	0
physical address	line number
physical address	line number
.	.
.	.
.	.
symbol index	0
physical address	line number
physical address	line number

The first entry in a function grouping has line number 0 and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries are relative to the beginning of the function and appear in increasing order of address.

Line number declaration

The structure declaration currently used for line number entries is shown below.

```
struct lineno
{
    union
    {
        long   l_symndx; /* symtbl index of func name */
        long   l_paddr;  /* paddr of line number */
    } l_addr;
    unsigned short l_lnno; /* line number */
};

#define LINENO    struct lineno
#define LINESZ    6
```

Symbol table

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the sequence shown in Table 7-11, "COFF symbol table" (page 145).

Table 7-11 COFF symbol table

filename 1
function 1
C_WEAKEXT aliases for function 1
function 1b (alias)
...
local symbols for function 1
function 2
C_WEAKEXT aliases for function 2
...
local symbols for function 2
...
statics
...
filename 2
function 1
C_WEAKEXT aliases for function 1
...
local symbols for function 1
...
statics
...
defined global symbols
undefined global symbols

The word “statics” in Table 7-11, “COFF symbol table” (this page) means symbols defined with the C language storage class static outside any function. The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the value, the type, and other information.

Special symbols

The symbol table contains some special symbols that are generated by **as** and other tools. These symbols are given in Table 7-12, “Special symbols in the symbol table” (this page).

Table 7-12 Special symbols in the symbol table

Symbol	Meaning
.file	filename
.text	address of .text section
.data	address of .data section
.bss	address of .bss section
.bb	address of start of inner block
.eb	address of end of inner block
.bf	address of start of function
.ef	address of end of function
.target	pointer to the structure or union returned by a function
.xfake	dummy tag name for structure, union, or enumeration
.eos	end of members of structure, union, or enumeration
etext	next available address after the end of the output section .text
edata	next available address after the end of the output section .data
end	next available address after the end of the output section .bss

Six of these special symbols occur in pairs. The **.bb** and **.eb** symbols indicate the boundaries of inner blocks; a **.bf** and **.ef** pair brackets each function. An **.xfake** and **.eos** pair names and defines the limit of structures, unions, and enumerations that were not named. The **.eos** symbol also appears after named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler invents a name to be used in the symbol table. The name chosen for the symbol table is **.xfake**, where *x* is an integer. If there are three unnamed structures, unions, or enumerations in the source, their tag names are **.0fake**, **.1fake**, and **.2fake**. Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entries.

Inner blocks

The C language defines a block as a compound statement that begins and ends with braces, “{” and “}”. An inner block is a block that occurs within a function (which is also a block).

For each inner block that has local symbols defined, a special symbol **.bb** is put in the symbol table immediately before the first local symbol of that block. Also a special symbol **.eb** is put in the symbol table immediately after the last local symbol of that block. The sequence is shown in Table 7-13, “Special symbols (.bb and .eb)” (this page).

Table 7-13 Special symbols (.bb and .eb)

.bb
local symbols for that block
.eb

Because inner blocks can be nested by several levels, the **.bb-.eb** pairs and associated symbols may also be nested as shown below:

```

{                               /* block 1 */
    int i;
    char c;
    ...
    {                           /* block 2 */
        long a;
        ...
        {                       /* block 3 */
            int x;
            ....
        }                       /* block 3 */
    }                           /* block 2 */
}                               /* block 1 */

{                               /* block 4 */
    long i;
    ...
}                               /* block 4 */

```

The symbol table would look like Table 7-14, “Example of the symbol table” (page 148)

Table 7-14 Example of the symbol table

.bb for block 1
i
c
.bb for block 2
a
.bb for block 3
x
.eb for block 3
.eb for block 2
.bb for block 4
i
.eb for block 4
.eb for block 1

Symbols and functions

For each function, a special symbol **.bf** is put between the function name and the first local symbol of the function in the symbol table. Also, a special symbol **.ef** is put immediately after the last local symbol of the function in the symbol table. The sequence is shown in Table 7-15, “Symbols for functions” (this page).

Table 7-15 Symbols for functions

function name
.bf
local symbol
.ef

Symbol table entries

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries each contain 18 bytes of information. The meaning of each of the fields in the symbol table entry is described in Table 7-16, “Symbol table entry format” (page 149). It should be noted that indices for symbol table entries begin at 0 and count upward. Each auxiliary entry also counts as one symbol.

Table 7-16 Symbol table entry format

Bytes	Declaration	Name	Description
0-7	(see text below)	<code>_n</code>	These 8 bytes contain either a symbol name or an index to a symbol
8-11	<code>long int</code>	<code>n_value</code>	Symbol value; storage class dependent
12-13	<code>short</code>	<code>n_scnun</code>	Section number of symbol
14-15	<code>unsigned short</code>	<code>n_type</code>	Basic and derived type specification
16	<code>char</code>	<code>n_sclass</code>	Storage class of symbol
17	<code>char</code>	<code>n_numaux</code>	Number of auxiliary entries

Symbol names

The first 8 bytes in the symbol table entry are a union of a character array and two longs. If the symbol name is eight characters or less, the (null-padded) symbol name is stored there. If the symbol name is longer than eight characters, then the entire symbol name is stored in the string table. In this case, the 8 bytes contain two long integers, the first is zero, and the second is the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first 4 bytes serve to distinguish a symbol table entry with an offset from one with a name in the first 8 bytes as shown in Table 7-17, “Name field” (this page).

Table 7-17 Name field

Bytes	Declaration	Name	Description
0-7	<code>char</code>	<code>n_name</code>	8-character null-padded symbol name
0-3	<code>long</code>	<code>n_zeroes</code>	Zero in this field indicates the name is in the string table
4-7	<code>long</code>	<code>n_offset</code>	Offset of the name in the string table

Special symbols generated by the C compilation system are discussed in “Special symbols” (page 146).

Storage classes

The storage class field has one of the values described in Table 7-18, “Storage classes” (this page) These #defines may be found in the header file *fstorclass.h*.

Table 7-18 Storage classes

Mnemonic	Value	Storage Class
C_EFCN	-1	physical end of a function
C_NULL	0	-
C_AUTO	1	automatic variable
C_EXT	2	external symbol
C_STAT	3	static
C_REG	4	register variable
C_EXTDEF	5	external definition
C_LABEL	6	label
C_ULABEL	7	undefined label
C_MOS	8	member of structure
C_ARG	9	function argument
C_STRTAG	10	structure tag
C_MOU	11	member of union
C_UNTAG	12	union tag
C_TPDEF	13	type definition
C_USTATIC	14	uninitialized static
C_ENTAG	15	enumeration tag
C_MOE	16	member of enumeration
C_REGPARM	17	register parameter
C_FIELD	18	bit field
C_BLOCK	100	beginning and end of block
C_FCN	101	beginning and end of function
C_EOS	102	end of structure
C_FILE	103	file name
C_LINE	104	used only by utility programs
C_ALIAS	105	duplicated tag
C_HIDDEN	106	like static, used to avoid name conflicts
C_SHADOW	107	shadow symbol
C_WEAKEXT	108	like C_EXT, but with weak linkage

All of these storage classes except for `C_ALIAS` and `C_HIDDEN` are generated by the `cc` or `as` commands. The `compress` utility, `cprcs(CP)`, generates the `C_ALIAS` mnemonic. This utility removes duplicated structure, union, and enumeration definitions and puts alias entries in their places. The storage class `C_HIDDEN` is not used by any UNIX system tools. Some of these storage classes are used only internally by the C compilation systems. These storage classes are `C_EFCN`, `C_EXTDEF`, `C_ULABEL`, `C_USTATIC`, and `C_LINE`.

Storage classes for special symbols

Some special symbols are restricted to certain storage classes. They are given in Table 7-19, “Storage class by special symbols” (this page).

Table 7-19 Storage class by special symbols

Special Symbol	Storage Class
<code>.file</code>	<code>C_FILE</code>
<code>.bb</code>	<code>C_BLOCK</code>
<code>.eb</code>	<code>C_BLOCK</code>
<code>.bf</code>	<code>C_FCN</code>
<code>.ef</code>	<code>C_FCN</code>
<code>.target</code>	<code>C_AUTO</code>
<code>.xfake</code>	<code>C_STRTAG</code> , <code>C_UNTAG</code> , <code>C_ENTAG</code>
<code>.eos</code>	<code>C_EOS</code>
<code>.text</code>	<code>C_STAT</code>
<code>.data</code>	<code>C_STAT</code>
<code>.bss</code>	<code>C_STAT</code>

Also some storage classes are used only for certain special symbols. They are summarized in Table 7-20, “Restricted storage classes” (this page).

Table 7-20 Restricted storage classes

Storage Class	Special Symbol
<code>C_BLOCK</code>	<code>.bb</code> , <code>.eb</code>
<code>C_FCN</code>	<code>.bf</code> , <code>.ef</code>
<code>C_EOS</code>	<code>.eos</code>
<code>C_FILE</code>	<code>.file</code>

Symbol value field

The meaning of the value of a symbol depends on its storage class. This relationship is summarized in Table 7-21, “Storage class and value” (this page).

Table 7-21 Storage class and value

Storage Class	Meaning of Value
C_AUTO	stack offset in bytes
C_EXT	relocatable address
C_STAT	relocatable address
C_REG	register number
C_LABEL	relocatable address
C_MOS	offset in bytes
C_ARG	stack offset in bytes
C_STRTAG	0
C_MOU	0
C_UNTAG	0
C_TPDEF	0
C_ENTAG	0
C_MOE	enumeration value
C_REGPARM	register number
C_FIELD	bit displacement
C_BLOCK	relocatable address
C_FCN	relocatable address
C_EOS	size
C_FILE	(see text below)
C_ALIAS	tag index
C_HIDDEN	relocatable address
C_WEAKEXT	relocatable address

If a symbol has storage class C_FILE, the value of that symbol equals the symbol table entry index of the next **.file** symbol. That is, the **.file** entries form a one-way linked list in the symbol table. If there are no more **.file** entries in the symbol table, the value of the symbol is the index of the first global symbol.

Relocatable symbols have a value equal to the virtual address of that symbol. When the section is relocated by the link editor, the value of these symbols changes.

Section number field

Section numbers are listed in Table 7-22, “Section number” (this page).

Table 7-22 Section number

Mnemonic	Section Number	Meaning
N_DEBUG	-2	Special symbolic debugging symbol
N_ABS	-1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1-077777	Section number where symbol is defined

A special section number (-2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the name of the file. A section number of -1 indicates that the symbol has a value but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and `.eos` symbols.

With one exception, a section number of 0 indicates a relocatable external symbol that is not defined in the current file. The one exception is a multiply-defined external symbol (i.e., FORTRAN common or an uninitialized variable-defined external to a function in C). In the symbol table of each file where the symbol is defined, the section number of the symbol is 0, and the value of the symbol is a positive number giving the size of the symbol. When the files are combined to form an executable object file, the link editor combines all the input symbols of the same name into one symbol with the section number of the `.bss` section. The maximum size of all the input symbols with the same name is used to allocate space for the symbol and the value becomes the address of the symbol. This is the only case where a symbol has a section number of 0 and a non-zero value.

Section numbers and storage classes

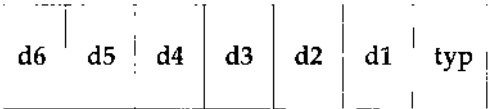
Symbols having certain storage classes are also restricted to certain section numbers. They are summarized in Table 7-23, “Section number and storage class” (this page).

Table 7-23 Section number and storage class

Storage Class	Section Number
C_AUTO	N_ABS
C_EXT	N_ABS, N_UNDEF, N_SCNUM
C_STAT	N_SCNUM
C_REG	N_ABS
C_LABEL	N_UNDEF, N_SCNUM
C_MOS	N_ABS
C_ARG	N_ABS
C_STRTAG	N_DEBUG
C_MOU	N_ABS
C_UNTAG	N_DEBUG
C_TPDEF	N_DEBUG
C_ENTAG	N_DEBUG
C_MOE	N_ABS
C_REGPARM	N_ABS
C_FIELD	N_ABS
C_BLOCK	N_SCNUM
C_FCN	N_SCNUM
C_EOS	N_ABS
C_FILE	N_DEBUG
C_ALIAS	N_DEBUG
C_WEAKEXT	N_ABS, N_UNDEF, N_SCNUM

Type entry

The type field in the symbol table entry contains information about the basic and derived type for the symbol. This information is generated by the C compilation system only if the `-g` option in the *Programmer's Reference Manual Volume 1* is used. Each symbol has exactly one basic or fundamental type but can have more than one derived type. The format of the 16-bit type entry is:



Bits 0 through 3, called **typ**, indicate one of the fundamental types given in Table 7-24, "Fundamental types" (this page).

Table 7-24 Fundamental types

Mnemonic	Value	Type
T_NULL	0	type not assigned
T_ARG	1	function argument (used only by compiler)
T_CHAR	2	character
T_SHORT	3	short integer
T_INT	4	integer
T_LONG	5	long integer
T_FLOAT	6	floating point
T_DOUBLE	7	double word
T_STRUCT	8	structure
T_UNION	9	union
T_ENUM	10	enumeration
T_MOE	11	member of enumeration
T_UCHAR	12	unsigned character
T_USHORT	13	unsigned short
T_UINT	14	unsigned integer
T_ULONG	15	unsigned long

Bits 4 through 15 are arranged as six 2-bit fields marked **d1** through **d6**. These **d** fields represent levels of the derived types given in Table 7-25, “Derived types” (this page).

Table 7-25 Derived types

Mnemonic	Value	Type
DT_NON	0	no derived type
DT_PTR	1	pointer
DT_FCN	2	function
DT_ARY	3	array

The following examples demonstrate the interpretation of the symbol table entry representing type:

```
char *func();
```

Here *func* is the name of a function that returns a pointer to a character. The fundamental type of *func* is 2 (character), the **d1** field is 2 (function), and the **d2** field is 1 (pointer). Therefore, the type word in the symbol table for *func* contains the hexadecimal number 0x62, which is interpreted to mean a function that returns a pointer to a character:

```
short *tabptr[10][25][3];
```

Here *tabptr* is a three-dimensional array of pointers to short integers. The fundamental type of *tabptr* is 3 (short integer); the **d1**, **d2**, and **d3** fields each contains a 3 (array), and the **d4** field is 1 (pointer). Therefore, the type entry in the symbol table contains the hexadecimal number 0x7f3 indicating a three-dimensional array of pointers to short integers.

Type entries and storage classes

Table 7-26, “Type entries by storage class” (this page) shows the type entries that are legal for each storage class.

Table 7-26 Type entries by storage class

Storage Class	d Entry			typ Entry Basic Type
	Function?	Array?	Pointer?	
C_AUTO	no	yes	yes	Any except T_MOE
C_EXT	yes	yes	yes	Any except T_MOE
C_STAT	yes	yes	yes	Any except T_MOE
C_REG	no	no	yes	Any except T_MOE
C_LABEL	no	no	no	T_NULL
C_MOS	no	yes	yes	Any except T_MOE
C_ARG	yes	no	yes	Any except T_MOE
C_STRTAG	no	no	no	T_STRUCT
C_MOU	no	yes	yes	Any except T_MOE
C_UNTAG	no	no	no	T_UNION
C_TPDEF	no	yes	yes	Any except T_MOE
C_ENTAG	no	no	no	T_ENUM
C_MOE	no	no	no	T_MOE
C_REGPARM	no	no	yes	Any except T_MOE
C_FIELD	no	no	no	T_ENUM, T_UCHAR, T_USHORT, T_UNIT, T_ULONG
C_BLOCK	no	no	no	T_NULL
C_FCN	no	no	no	T_NULL
C_EOS	no	no	no	T_NULL
C_FILE	no	no	no	T_NULL
C_ALIAS	no	no	no	T_STRUCT, T_UNION, T_ENUM
C_WEAKEXT	no	no	no	Any except T_MOE

Conditions for the **d** entries apply to **d1** through **d6**, except that it is impossible to have two consecutive derived types of function.

Although function arguments can be declared as arrays, they are changed to pointers by default. Therefore, no function argument can have array as its first derived type.

Structure for symbol table entries

The C language structure declaration for the symbol table entry is shown below. This declaration may be found in the header file *syms.h*.

```

struct symtent
{
    union
    {
        char            _n_name[SYMNMLEN]; /* symbol name*/
        struct
        {
            long        _n_zeroes;          /* symbol name */
            long        _n_offset;          /* location in string table */
        } _n_n;
        char            *_n_nptr[2];        /* allows overlaying */
    } _n;
    unsigned long       n_value;            /* value of symbol */
    short               n_scnun;            /* section number */
    unsigned short      n_type;             /* type and derived */
    char                n_sclass;           /* storage class */
    char                n_numaux;           /* number of aux entries */
};

#define n_name          _n._n_name
#define n_zeroes        _n._n_n._n_zeroes
#define n_offset        _n._n_n._n_offset
#define n_nptr          _n._n_nptr[1]

#define SYMNMLEN 8
#define SYMESZ 18 /* size of a symbol table entry */

```

Auxiliary table entries

An auxiliary table entry of a symbol contains the same number of bytes as the symbol table entry. However, unlike symbol table entries, the format of an auxiliary table entry of a symbol depends on its type and storage class. They are summarized in Table 7-27, “Auxiliary symbol table entries” (this page).

Table 7-27 Auxiliary symbol table entries

Name	Storage Class	Type Entry		Auxiliary Entry Format
		d1	typ	
.file	C_FILE	DT_NON	T_NULL	file name
.text, .data,	C_STAT	DT_NON	T_NULL	section
.bss				
tagname	C_STRTAG	DT_NON	T_NULL	tag name
	C_UNTAG			
	C_ENTAG			
.eos	C_EOS	DT_NON	T_NULL	end of structure
fname	C_EXT	DT_FCN	(Note 1)	function
	C_STAT			
	C_WEAKEXT			
arrname	(Note 2)	DT_ARY	(Note 1)	array
.bb, .eb	C_BLOCK	DT_NON	T_NULL	beginning and end of block
.bf, .ef	C_FCN	DT_NON	T_NULL	beginning and end of function
name related to structure, union, enumeration	(Note 2)	DT_PTR, DT_ARR, DT_NON	T_STRUCT, T_UNION, T_ENUM	name related to structure, union, enumeration

Notes to Table 7-27, “Auxiliary symbol table entries” (page 159):

1. Any except T_MOE.
2. C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF.

In Table 7-27, “Auxiliary symbol table entries” (page 159), *tagname* means any symbol name including the special symbol *.xfake*, and *fcname* and *arrname* represent any symbol name for a function or an array respectively. Any symbol that satisfies more than one condition in Table 7-27, “Auxiliary symbol table entries” (page 159) should have a union format in its auxiliary entry.

NOTE It is a mistake to assume how many auxiliary entries are associated with any given symbol table entry. This information is available and should be obtained from the `n_numaux` field in the symbol table.

File names

Each of the auxiliary table entries for a file name contains a 14-character file name in bytes 0 through 13. The remaining bytes are zeroes.

Sections

The auxiliary table entries for sections have the format as shown in Table 7-28, “Format for auxiliary table entries for sections” (this page).

Table 7-28 Format for auxiliary table entries for sections

Bytes	Declaration	Name	Description
0-3	long int	<code>x_scnlen</code>	section length
4-5	unsigned short	<code>x_nreloc</code>	number of relocation entries
6-7	unsigned short	<code>x_nlinno</code>	number of line numbers
8-17	-	-	unused (filled with zeroes)

Tag names

The auxiliary table entries for tag names have the format shown in Table 7-29, “Tag names table entries” (this page).

Table 7-29 Tag names table entries

Bytes	Declaration	Name	Description
0-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of structure, union, and enumeration
8-11	-	-	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry beyond this structure, union, or enumeration
16-17	-	-	unused (filled with zeroes)

End of structures

The auxiliary table entries for the end of structures have the format shown in Table 7-30, “Table entries for end of structures” (this page):

Table 7-30 Table entries for end of structures

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of structure, union, or enumeration
8-17	-	-	unused (filled with zeroes)

Functions

The auxiliary table entries for functions have the format shown in Table 7-31, “Table entries for functions” (this page):

Table 7-31 Table entries for functions

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-7	long int	x_fsize	size of function (in bytes)
8-11	long int	x_lnnoptr	file pointer to line number
12-15	long int	x_endndx	index of next entry beyond this point
16-17	unsigned short	x_tvndx	index of function’s address in the transfer vector table (not used in the UNIX system)

Arrays

The auxiliary table entries for arrays have the format shown in Table 7-32, “Table entries for arrays” (this page). Defining arrays having more than four dimensions produces a warning message.

Table 7-32 Table entries for arrays

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	unsigned short	x_lnnno	line number of declaration
6-7	unsigned short	x_size	size of array
8-9	unsigned short	x_dimen[0]	first dimension
10-11	unsigned short	x_dimen[1]	second dimension
12-13	unsigned short	x_dimen[2]	third dimension
14-15	unsigned short	x_dimen[3]	fourth dimension
16-17	-	-	unused (filled with zeroes)

End of blocks and functions

The auxiliary table entries for the end of blocks and functions have the format shown in Table 7-33, “End of block and function entries” (this page):

Table 7-33 End of block and function entries

Bytes	Declaration	Name	Description
0-3	-	-	unused (filled with zeroes)
4-5	unsigned short	x_inno	C-source line number
6-17	-	-	unused (filled with zeroes)

Beginning of blocks and functions

The auxiliary table entries for the beginning of blocks and functions have the format shown in Table 7-34, “Format for beginning of block and function” (this page):

Table 7-34 Format for beginning of block and function

Bytes	Declaration	Name	Description
0-3	-	-	unused (filled with zeroes)
4-5	unsigned short	x_inno	C-source line number
6-11	-	-	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry past this block
16-17	-	-	unused (filled with zeroes)

Names related to structures, unions, and enumerations

The auxiliary table entries for structure, union, and enumeration symbols have the format shown in Table 7-35, “Entries for structures, unions, and enumerations” (this page):

Table 7-35 Entries for structures, unions, and enumerations

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of the structure, union, or enumeration
8-17	-	-	unused (filled with zeroes)

Aggregates defined by typedef may or may not have auxiliary table entries. For example,

```
typedef struct people STUDENT;

struct people
{
    char name[20];
    long id;
};

typedef struct people EMPLOYEE;
```

The symbol EMPLOYEE has an auxiliary table entry in the symbol table, but symbol STUDENT will not because it is a forward reference to a structure.

Auxiliary entry declaration

The C language structure declaration for an auxiliary symbol table entry is shown below. This declaration may be found in the header file *syms.h*.

```

union auxent
{
    struct
    {
        long    x_tagndx;
        union
        {
            struct
            {
                unsigned short  x_lnno;
                unsigned short  x_size;
            } x_lnsz;
            long    x_fsize;
        } x_misc;
        union
        {
            struct
            {
                long    x_lnnoptr;
                long    x_endndx;
            } x_fcn;
            struct
            {
                unsigned short  x_dimen[DIMNUM];
            } x_ary;
        } x_fcary;
        unsigned short  x_tvndx;
    } x_sym;
    struct
    {
        char    x_fname[FILNMLEN];
    } x_file;
    struct
    {
        long    x_scnlen;
        unsigned short  x_nreloc;
        unsigned short  x_nlinno;
    } x_scn;
    struct
    {
        long    x_tvfill;
        unsigned short  x_tvlen;
        unsigned short  x_tvran[2];
    } x_tv;
}

#define FILNMLEN 14
#define DIMNUM 4
#define AUXENT union auxent
#define AUXESZ 18

```

String table

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes; offsets into the string table, therefore, are greater than or equal to 4. For example, given a file containing two symbols (with names longer than eight characters, *long_name_1* and *another_one*) the string table has the format as shown in Table 7-36, “String table” (this page):

Table 7-36 String table

'l'	'o'	'n'	'g'
'_'	'n'	'a'	'm'
'e'	'_'	'l'	'\0'
'a'	'n'	'o'	't'
'h'	'e'	'r'	'_'
'o'	'n'	'e'	'\0'

The index of *long_name_1* in the string table is 4 and the index of *another_one* is 16.

Access routines

Access routines are used for reading the various parts of a common object file. Although the calling program must know the detailed structure of the parts of the object file it processes, the routines effectively insulate the calling program from the knowledge of the overall structure of the object file.

The access routines can be divided into four categories:

1. functions that open or close an object file
2. functions that read header or symbol table information
3. functions that position an object file at the start of a particular section of the object file
4. a function that returns the symbol table index for a particular symbol

These routines can be found in the library *libld.a* and are listed in the *Programmer's Reference*. A summary of what is available can be found in **ldfcn(FP)**.

Chapter 8

ELF object files

This chapter describes the executable and linking format (ELF) of the object files produced by the C compilation system. The first section, “Program linking” (page 171), focuses on how the format pertains to building programs. The second section, “Program execution” (page 194), focuses on how the format pertains to loading programs. See “Link editing” in the *Programming Tools Guide* for more information.

There are three main types of object files:

- A *relocatable file* holds code and data suitable for linking with other object files to create an executable or a shared object file.
- An *executable file* holds a program suitable for execution; the file specifies how `exec()` creates a program’s process image.
- A *shared object file* holds code and data suitable for linking in two contexts. First, the link editor processes the shared object file with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image.

Programs manipulate object files with the functions contained in the ELF access library, `libelf`. See `elf(5)` for more information.

File format

The ELF object file format provides two views of a file’s contents:

1. a view used during program linking
2. a view used during program execution

Figure 8-1, “Object file format” shows an object file’s organization.

Linking view		Execution view	
ELF header		ELF header	
Program header table (optional)		Program header table	
Section 1		Segment 1	
...			
Section n		Segment 2	
...			
...		...	
Section header table		Section header table (optional)	

Figure 8-1 Object file format

Common to both views is an ELF header that resides at the beginning and holds a “road map” describing the file’s organization.

In the linking view, sections hold the bulk of object file information which includes instructions, data, symbol table, and relocation information.

A program header table tells the system how to create a process image from the ELF object. In the program execution view, files used to build a process image (execute a program) must have a program header table; relocatable files do not need one.

A section header table contains information describing the file’s sections. Every section has an entry in the table; each entry gives information such as the section name, the section size, and so forth. Files used during linking must have a section header table; other object files may or may not have one.

NOTE Although the figure shows the program header table immediately after the ELF header, and the section header table following the sections, actual files may differ. Moreover, sections and segments have no specified order. Only the ELF header has a fixed position in the file.

Data representation

As described here, the object file format supports various processors with 8-bit bytes and 32-bit architectures. Nevertheless, it is intended to be extensible to larger (or smaller) architectures. Object files therefore represent some control data with a machine-independent format, making it possible to identify object files and interpret their contents in a common way. Remaining data in an object file use the encoding of the target processor, regardless of the machine on which the file was created.

Table 8-1 32-bit data types

Name	Size	Alignment	Purpose
Elf32_Addr	4	4	Unsigned program address
Elf32_Half	2	2	Unsigned medium integer
Elf32_Off	4	4	Unsigned file offset
Elf32_Sword	4	4	Signed large integer
Elf32_Word	4	4	Unsigned large integer
unsigned char	1	1	Unsigned small integer

All data structures that the object file format defines follow the “natural” size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 4-byte alignment for 4-byte objects, to force structure sizes to a multiple of 4, and so forth. Data also have suitable alignment from the beginning of the file. Thus, for example, a structure containing an `Elf32_Addr` member will be aligned on a 4-byte boundary within the file. For portability reasons, ELF uses no bit-fields.

Program linking

This section describes the object file information and system actions that create static program representations from relocatable files and shared objects.

ELF header

Some object file control structures can grow, because the ELF header contains their actual sizes. If the object file format changes, a program may encounter control structures that are larger or smaller than expected. Programs might therefore ignore “extra” information. The treatment of “missing” information depends on context and will be specified when and if extensions are defined.

Example 8-1 ELF header

```

#define EI_NIDENT      16

typedef struct {
    unsigned char  e_ident[EI_NIDENT];
    Elf32_Half     e_type;
    Elf32_Half     e_machine;
    Elf32_Word     e_version;
    Elf32_Addr     e_entry;
    Elf32_Off      e_phoff;
    Elf32_Off      e_shoff;
    Elf32_Word     e_flags;
    Elf32_Half     e_ehsize;
    Elf32_Half     e_phentsize;
    Elf32_Half     e_phnum;
    Elf32_Half     e_shentsize;
    Elf32_Half     e_shnum;
    Elf32_Half     e_shstrndx;
} Elf32_Ehdr;

```

e_ident The initial bytes mark the file as an object file and provide machine-independent data with which to decode and interpret the file's contents. Complete descriptions appear in "ELF identification" (page 174).

e_type This member identifies the object file type.

Name	Value	Meaning
ET_NONE	0	No file type
ET_REL	1	Relocatable file
ET_EXEC	2	Executable file
ET_DYN	3	Shared object file
ET_CORE	4	Core file
ET_LOPROC	0xff00	Processor-specific
ET_HIPROC	0xffff	Processor-specific

Although the core file contents are unspecified in the System V Application Binary Interface, type **ET_CORE** is reserved to mark the file. Values from **ET_LOPROC** through **ET_HIPROC** (inclusive) are reserved for processor-specific semantics. Other values are reserved and will be assigned to new object file types as necessary.

`e_machine` This member's value specifies the required architecture for an individual file.

Name	Value	Meaning
EM_NONE	0	No machine
EM_M32	1	AT&T WE 32100
EM_SPARC	2	SPARC
EM_386	3	Intel386™ CPU
EM_68K	4	Motorola 68000
EM_88K	5	Motorola 88000
EM_860	7	Intel860™ CPU
EM_MIPS	8	MIPS R2000

Other values are reserved and will be assigned to new machines as necessary. Processor-specific ELF names use the machine name to distinguish them. For example, the flags mentioned in the next table use the prefix `EF_`.

`e_version` This member identifies the object file version.

Name	Value	Meaning
EV_NONE	0	Invalid version
EV_CURRENT	1	Current version

The value 1 signifies the original file format; extensions will create new versions with higher numbers. Although the value of `EV_CURRENT` is shown as 1 in the previous table, it will change as necessary to reflect the current version number.

`e_entry` This member gives the virtual address to which the system first transfers control, thus starting the process. If the file has no associated entry point, this member holds zero.

`e_phoff` This member holds the program header table's file offset in bytes. If the file has no program header table, this member holds zero.

`e_shoff` This member holds the section header table's file offset in bytes. If the file has no section header table, this member holds zero.

`e_flags` This member holds processor-specific flags associated with the file. Flag names take the form `EF_machine_flag`.

|| **NOTE** See “ELF header flags” (page 177) for flag definitions.

`e_ehsize` This member holds the ELF header’s size in bytes.

`e_phentsize` This member holds the size in bytes of one entry in the file’s program header table; all entries are the same size.

`e_phnum` This member holds the number of entries in the program header table. Thus the product of `e_phentsize` and `e_phnum` gives the table’s size in bytes. If a file has no program header table, `e_phnum` holds the value zero.

`e_shentsize` This member holds a section header’s size in bytes. A section header is one entry in the section header table; all entries are the same size.

`e_shnum` This member holds the number of entries in the section header table. Thus the product of `e_shentsize` and `e_shnum` gives the section header table’s size in bytes. If a file has no section header table, `e_shnum` holds the value zero.

`e_shstrndx` This member holds the section header table index of the entry associated with the section name string table. If the file has no section name string table, this member holds the value `SHN_UNDEF`.

|| **NOTE** See “Section header” (page 177) and “String table” (page 187) for more information.

ELF identification

ELF provides an object file framework to support multiple processors, multiple data encodings, and multiple classes of machines. To support this object file family, the initial bytes of the file specify how to interpret the file, independent of the processor on which the inquiry is made and independent of the file’s remaining contents.

The initial bytes of an ELF header (and an object file) correspond to the `e_ident` member.

Table 8-2 `e_ident[]` identification indexes

Name	Value	Purpose
EI_MAG0	0	File identification
EI_MAG1	1	File identification
EI_MAG2	2	File identification
EI_MAG3	3	File identification
EI_CLASS	4	File class
EI_DATA	5	Data encoding
EI_VERSION	6	File version
EI_PAD	7	Start of padding bytes
EI_NIDENT	16	Size of <code>e_ident[]</code>

These indexes access bytes that hold the following values.

EI_MAG0 to EI_MAG3

A file's first 4 bytes hold a "magic number", identifying the file as an ELF object file.

Name	Value	Position
ELFMAG0	0x7f	<code>e_ident[EI_MAG0]</code>
ELFMAG1	'E'	<code>e_ident[EI_MAG1]</code>
ELFMAG2	'L'	<code>e_ident[EI_MAG2]</code>
ELFMAG3	'F'	<code>e_ident[EI_MAG3]</code>

EI_CLASS

The next byte, `e_ident[EI_CLASS]`, identifies the file's class, or capacity.

Name	Value	Meaning
ELFCLASSNONE	0	Invalid class
ELFCLASS32	1	32-bit objects
ELFCLASS64	2	64-bit objects

The file format is designed to be portable among machines of various sizes, without imposing the sizes of the largest machine on the smallest. Class **ELFCLASS32** supports machines with files and virtual address spaces up to 4 gigabytes and uses the basic types defined in the previous table.

Class **ELFCLASS64** is reserved for 64-bit architectures. Its appearance here shows how the object file may change, but the 64-bit format is otherwise unspecified. Other classes will be defined as necessary, with different basic types and sizes for object file data.

EI_DATA Byte `e_ident[EI_DATA]` specifies the data encoding of the processor-specific data in the object file. The following encodings are currently defined.

Name	Value	Meaning
ELFDATANONE	0	Invalid data encoding
ELFDATA2LSB	1	See below
ELFDATA2MSB	2	See below

Other values are reserved and will be assigned to new encodings as necessary.

EI_VERSION Byte `e_ident[EI_VERSION]` specifies the ELF header version number. Currently, this value must be **EV_CURRENT**, for `e_version`.

EI_PAD This value marks the beginning of the unused bytes in `e_ident`. These bytes are reserved and set to zero; programs that read object files should ignore them. The value of **EI_PAD** will change in the future if currently unused bytes are given meanings.

A file's data encoding specifies how to interpret the basic objects in a file. Class **ELFCLASS32** files use objects that occupy 1, 2, and 4 bytes. Under the defined encodings, objects are represented as detailed in Figure 8-2, "Data encoding **ELFDATA2LSB**" (this page). Byte numbers appear in the upper left corners.

Encoding **ELFDATA2LSB** specifies 2's complement values, with the least significant byte occupying the lowest address.

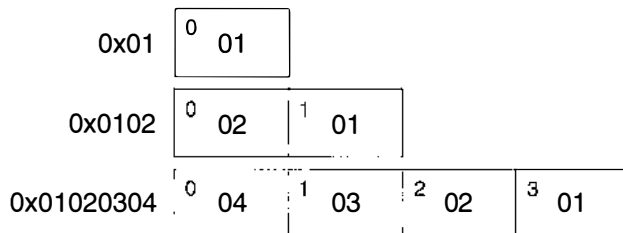


Figure 8-2 Data encoding **ELFDATA2LSB**

Encoding **ELFDATA2MSB** specifies 2's complement values, with the most significant byte occupying the lowest address.

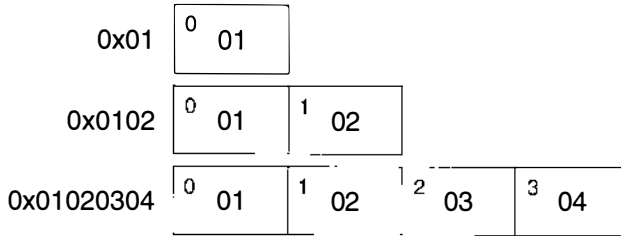


Figure 8-3 Data encoding ELFDATA2MSB

ELF header flags

The ELF header's `e_flags` member holds bit flags associated with the file.

Section header

An object file's section header table lets one locate all the file's sections. The section header table is an array of **Elf32_Shdr** structures as described in Table 8-3, "Special section indexes" (this page). A section header table index is a subscript into this array. The ELF header's `e_shoff` member gives the byte offset from the beginning of the file to the section header table. `e_shnum` tells how many entries the section header table contains. `e_shentsize` gives the size in bytes of each entry.

Some section header table indexes are reserved; an object file will not have sections for these special indexes.

Table 8-3 Special section indexes

Name	Value
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00
SHN_HIPROC	0xff1f
SHN_ABS	0xfff1
SHN_COMMON	0xfff2
SHN_HIRESERVE	0xffff

SHN_UNDEF This value marks an undefined, missing, irrelevant, or otherwise meaningless section reference. For example, a symbol “defined” relative to section number **SHN_UNDEF** is an undefined symbol.

NOTE Although index 0 is reserved as the undefined value, the section header table contains an entry for index 0. If the `e_shnum` member of the ELF header says a file has 6 entries in the section header table, they have the indexes 0 through 5.

SHN_LORESERVE This value specifies the lower bound of the range of reserved indexes.

SHN_LOPROC through **SHN_HIPROC**
Values in this inclusive range are reserved for processor-specific semantics.

SHN_ABS This value specifies absolute values for the corresponding reference. For example, symbols defined relative to section number **SHN_ABS** have absolute values and are not affected by relocation.

SHN_COMMON Symbols defined relative to this section are common symbols, such as FORTRAN, COMMON or unallocated C external variables.

SHN_HIRESERVE This value specifies the upper bound of the range of reserved indexes. The system reserves indexes between **SHN_LORESERVE** and **SHN_HIRESERVE**, inclusive; the values do not reference the section header table. The section header table does not contain entries for the reserved indexes.

Sections contain all information in an object file except the ELF header, the program header table, and the section header table. Moreover, object files’ sections satisfy several conditions.

- Every section in an object file has exactly one section header describing it. Section headers may exist that do not have a section.
- Each section occupies one contiguous (possibly empty) sequence of bytes within a file.
- Sections in a file may not overlap. No byte in a file resides in more than one section.
- An object file may have inactive space. The various headers and the sections might not “cover” every byte in an object file. The contents of the inactive data are unspecified.

A section header has the following structure.

```
typedef struct {
    Elf32_Word    sh_name;
    Elf32_Word    sh_type;
    Elf32_Word    sh_flags;
    Elf32_Addr    sh_addr;
    Elf32_Off     sh_offset;
    Elf32_Word    sh_size;
    Elf32_Word    sh_link;
    Elf32_Word    sh_info;
    Elf32_Word    sh_addralign;
    Elf32_Word    sh_entsize;
} Elf32_Shdr;
```

sh_name This member specifies the name of the section. Its value is an index into the section header string table section, giving the location of a null-terminated string.

NOTE See “String table” (page 187) for more information.

sh_type This member categorizes the section’s contents and semantics. Section types and their descriptions appear in Table 8-4, “Section types, sh_type” (page 180).

sh_flags Sections support 1-bit flags that describe miscellaneous attributes. Flag definitions appear in Table 8-4, “Section types, sh_type” (page 180).

sh_addr If the section will appear in the memory image of a process, this member gives the address at which the section’s first byte should reside. Otherwise, the member contains 0.

sh_offset This member’s value gives the byte offset from the beginning of the file to the first byte in the section. One section type, **SHT_NOBITS**, described in Table 8-4 (page 180), occupies no space in the file, and its **sh_offset** member locates the conceptual placement in the file.

<code>sh_size</code>	This member gives the section's size in bytes. Unless the section type is SHT_NOBITS , the section occupies <code>sh_size</code> bytes in the file. A section of type SHT_NOBITS may have a non-zero size, but it occupies no space in the file.
<code>sh_link</code>	This member holds a section header table index link, whose interpretation depends on the section type. Table 8-4, "Section types, <code>sh_type</code> " (this page) describes the values.
<code>sh_info</code>	This member holds extra information, whose interpretation depends on the section type. Table 8-4, "Section types, <code>sh_type</code> " (this page) describes the values.
<code>sh_addralign</code>	Some sections have address alignment constraints. For example, if a section holds a doubleword, the system must ensure doubleword alignment for the entire section. The value of <code>sh_addr</code> must be congruent to 0, modulo the value of <code>sh_addralign</code> . Currently, only 0 and positive integral powers of two are allowed. Values 0 and 1 mean the section has no alignment constraints.
<code>sh_entsize</code>	Some sections hold a table of fixed-size entries, such as a symbol table. For such a section, this member gives the size in bytes of each entry. The member contains 0 if the section does not hold a table of fixed-size entries.

A section header's `sh_type` member specifies the section's semantics.

Table 8-4 Section types, `sh_type`

Name	Value
SHT_NULL	0
SHT_PROGBITS	1
SHT_SYMTAB	2
SHT_STRTAB	3
SHT_RELA	4
SHT_HASH	5
SHT_DYNAMIC	6
SHT_NOTE	7
SHT_NOBITS	8
SHT_REL	9
SHT_SHLIB	10
SHT_DYNSYM	11
SHT_LOPROC	0x70000000
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
SHT_HIUSER	0xffffffff

- SHT_NULL** This value marks the section header as inactive; it does not have an associated section. Other members of the section header have undefined values.
- SHT_PROGBITS** The section holds information defined by the program, whose format and meaning are determined solely by the program.
- SHT_SYMTAB** and **SHT_DYNSYM**
 These sections hold a symbol table. Currently, an object file may have only one section of each type, but this restriction may be relaxed in the future. Typically, **SHT_SYMTAB** provides symbols for link editing, though it may also be used for dynamic linking. As a complete symbol table, it may contain many symbols unnecessary for dynamic linking. Consequently, an object file may also contain a **SHT_DYNSYM** section, which holds a minimal set of dynamic linking symbols, to save space.
- NOTE** See “Symbol table” (page 187) for details.
- SHT_STRTAB** The section holds a string table. An object file may have multiple string table sections.
- NOTE** See “String table” (page 187) for details.
- SHT_RELA** The section holds relocation entries with explicit addends, such as type **Elf32_Rela** for the 32-bit class of object files. An object file may have multiple relocation sections.
- NOTE** See “Relocation” (page 192) for details.
- SHT_HASH** The section holds a symbol hash table. Currently, an object file may have only one hash table, but this restriction may be relaxed in the future. See “Hash table” (page 212) in the second part of this chapter for details.
- SHT_DYNAMIC** The section holds information for dynamic linking. Currently, an object file may have only one dynamic section, but this restriction may be relaxed in the future. See “Dynamic section” (page 206) in the second part of this chapter for details.
- SHT_NOTE** The section holds information that marks the file in some way. See “Note section” (page 199) in the second part of this chapter for details.

- SHT_NOBITS** A section of this type occupies no space in the file but otherwise resembles **SHT_PROGBITS**. Although this section contains no bytes, the `sh_offset` member contains the conceptual file offset.
- SHT_REL** The section holds relocation entries without explicit addends, such as type **Elf32_Rel** for the 32-bit class of object files. An object file may have multiple relocation sections.

NOTE See “Relocation” (page 192) for details.

- SHT_SHLIB** This section type is reserved but has unspecified semantics.
- SHT_LOPROC** through **SHT_HIPROC**
Values in this inclusive range are reserved for processor-specific semantics.
- SHT_LOUSER** This value specifies the lower bound of the range of indexes reserved for application programs.
- SHT_HIUSER** This value specifies the upper bound of the range of indexes reserved for application programs. Section types between **SHT_LOUSER** and **SHT_HIUSER** may be used by the application, without conflicting with current or future system-defined section types.

Other section type values are reserved. As mentioned before, the section header for index 0 (**SHN_UNDEF**) exists, even though the index marks undefined section references. This entry holds the following.

Table 8-5 Section header table entry: index 0

Name	Value	Note
<code>sh_name</code>	0	No name
<code>sh_type</code>	SHT_NULL	Inactive
<code>sh_flags</code>	0	No flags
<code>sh_addr</code>	0	No address
<code>sh_offset</code>	0	No file offset
<code>sh_size</code>	0	No size
<code>sh_link</code>	SHN_UNDEF	No link information
<code>sh_info</code>	0	No auxiliary information
<code>sh_addralign</code>	0	No alignment
<code>sh_entsize</code>	0	No entries

A section header's `sh_flags` member holds 1-bit flags that describe the section's attributes. Defined values appear in Table 8-6, “Section attribute flags, `sh_flags`” (page 183); other values are reserved.

Table 8-6 Section attribute flags, sh_flags

Name	Value
SHF_WRITE	0x1
SHF_ALLOC	0x2
SHF_EXECINSTR	0x4
SHF_MASKPROC	0xf0000000

If a flag bit is set in `sh_flags`, the attribute is *on* for the section. Otherwise, the attribute is *off* or does not apply. Undefined attributes are set to zero.

SHF_WRITE The section contains data that should be writable during process execution.

SHF_ALLOC The section occupies memory during process execution. Some control sections do not reside in the memory image of an object file; this attribute is off for those sections.

SHF_EXECINSTR The section contains executable machine instructions.

SHF_MASKPROC All bits included in this mask are reserved for processor-specific semantics.

Two members in the section header, `sh_link` and `sh_info`, hold special information, depending on section type.

Table 8-7 sh_link and sh_info interpretation

sh_type	sh_link	sh_info
SHT_DYNAMIC	The section header index of the string table used by entries in the section.	0
SHT_HASH	The section header index of the symbol table to which the hash table applies.	0
SHT_REL SHT_RELA	The section header index of the associated symbol table.	The section header index of the section to which the relocation applies.
SHT_SYMTAB SHT_DYNSYM	The section header index of the associated string table.	One greater than the symbol table index of the last local symbol (binding STB_LOCAL).
other	SHN_UNDEF	0

Special sections

Various sections hold program and control information. Table 8-8, “Special sections” (this page) shows sections that are used by the system and have the indicated types and attributes.

Table 8-8 Special sections

Name	Type	Attributes
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	none
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.debug	SHT_PROGBITS	none
.dynamic	SHT_DYNAMIC	see below
.dynstr	SHT_STRTAB	SHF_ALLOC
.dysym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.got	SHT_PROGBITS	see below
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.interp	SHT_PROGBITS	none
.line	SHT_PROGBITS	none
.note	SHT_NOTE	none
.plt	SHT_PROGBITS	see below
.relname	SHT_REL	see below
.relaname	SHT_RELA	see below
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	none
.strtab	SHT_STRTAB	see below
.symtab	SHT_SYMTAB	see below
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

.bss This section holds uninitialized data that contribute to the program’s memory image. By definition, the system initializes the data with zeros when the program begins to run. The section occupies no file space, as indicated by the section type, **SHT_NOBITS**.

.comment This section holds version control information.

.data and **.data1**

These sections hold initialized data that contribute to the program’s memory image.

<code>.debug</code>	This section holds information for symbolic debugging. The contents are unspecified.
<code>.dynamic</code>	This section holds dynamic linking information. The section's attributes will include the <code>SHF_ALLOC</code> bit. Whether the <code>SHF_WRITE</code> bit is set is processor specific. See the second part of this chapter for more information.
<code>.dynstr</code>	This section holds strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries. See the second part of this chapter for more information.
<code>.dynsym</code>	This section holds the dynamic linking symbol table, as described in "Symbol table" (page 187). See the second part of this chapter for more information.
<code>.fini</code>	This section holds executable instructions that contribute to the process termination code. When a program exits normally, the system arranges to execute the code in this section.
<code>.got</code>	This section holds the global offset table.
<code>.hash</code>	This section holds a symbol hash table. See "Hash table" (page 212) in the second part of this chapter for more information.
<code>.init</code>	This section holds executable instructions that contribute to the process initialization code. When a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called <code>main</code> in C programs).
<code>.interp</code>	This section holds the path name of a program interpreter. See "Program interpreter" (page 204) for more information.
<code>.line</code>	This section holds line number information for symbolic debugging, which describes the correspondence between the source program and the machine code. The contents are unspecified.
<code>.note</code>	This section holds information in the format described in "Note section" (page 199).
<code>.plt</code>	This section holds the procedure linkage table.

.rel*name* and **.rel***a**name*

These sections hold relocation information, as described in “Relocation” (page 192). If the file has a loadable segment that includes relocation, the sections’ attributes will include the SHF_ALLOC bit; otherwise, that bit will be off. Conventionally, *name* is supplied by the section to which the relocations apply. Thus a relocation section for *.text* normally would have the name *.rel.text* or *.rela.text*.

.rodata and **.rodata1**

These sections hold read-only data that typically contribute to a non-writable segment in the process image. See “Program header” (page 194) in the second part of this chapter for more information.

.shstrtab This section holds section names.**.strtab** This section holds strings, most commonly the strings that represent the names associated with symbol table entries. If the file has a loadable segment that includes the symbol string table, the section’s attributes will include the SHF_ALLOC bit; otherwise, that bit will be off.**.symtab** This section holds a symbol table, as described in “Symbol table” (page 187). If the file has a loadable segment that includes the symbol table, the section’s attributes will include the SHF_ALLOC bit; otherwise, that bit will be off.**.text** This section holds the “text”, or executable instructions, of a program.

Section names with a dot (.) prefix are reserved for the system, although applications may use these sections if their existing meanings are satisfactory. Applications may use names without the prefix to avoid conflicts with system sections. The object file format lets one define sections not shown in the previous list. An object file may have more than one section with the same name.

Section names reserved for a processor architecture are formed by placing an abbreviation of the architecture name ahead of the section name. The name should be taken from the architecture names used for *e_machine*. For instance *.FOO.psect* is the *psect* section defined by the *FOO* architecture. Existing extensions are called by their historical names.

String table

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. One references a string as an index into the string table section. The first byte, which is index zero, is defined to hold a null character. Likewise, a string table's last byte is defined to hold a null character, ensuring null termination for all strings. A string whose index is zero specifies either no name or a null name, depending on the context. An empty string table section is permitted; its section header's `sh_size` member would contain zero. Non-zero indexes are invalid for an empty string table.

A section header's `sh_name` member holds an index into the section header string table section, as designated by the `e_shstrndx` member of the ELF header. The following figures show a string table with 25 bytes and the strings associated with various indexes.

Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

Figure 8-5 String table

Table 8-9 String table indexes

Index	String
0	none
1	name.
7	Variable
11	able
16	able
24	null string

As the example shows, a string table index may refer to any byte in the section. A string may appear more than once; references to substrings may exist; and a single string may be referenced multiple times. Unreferenced strings also are allowed.

Symbol table

An object file's symbol table holds information needed to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 both designates the first entry in the table and serves as the undefined symbol index.

Name	Value
STN_UNDEF	0

A symbol table entry has the following format.

```
typedef struct {
    Elf32_Word    st_name;
    Elf32_Addr    st_value;
    Elf32_Word    st_size;
    unsigned char st_info;
    unsigned char st_other;
    Elf32_Half    st_shndx;
} Elf32_Sym;
```

st_name This member holds an index into the object file's symbol string table, which holds the character representations of the symbol names. If the value is non-zero, it represents a string table index that gives the symbol name. Otherwise, the symbol table entry has no name.

NOTE External C symbols have the same names in C and object files' symbol tables.

st_value This member gives the value of the associated symbol. Depending on the context, this may be an absolute value, or an address.

st_size Many symbols have associated sizes. For example, a data object's size is the number of bytes contained in the object. This member holds 0 if the symbol has no size or an unknown size.

st_info This member specifies the symbol's type and binding attributes. A list of the values and meanings appears below. The following code shows how to manipulate the values.

```
#define ELF32_ST_BIND(i)    ((i)>>4)
#define ELF32_ST_TYPE(i)    ((i)&0xf)
#define ELF32_ST_INFO(b,t) (((b)<<4)+((t)&0xf))
```

st_other This member currently holds 0 and has no defined meaning.

st_shndx Every symbol table entry is *defined* in relation to some section. This member holds the relevant section header table index. Some section indexes indicate special meanings.

A symbol's binding determines the linkage visibility and behavior.

Table 8-10 Symbol binding, ELF32_ST_BIND

Name	Value
STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOPROC	13
STB_HIPROC	15

STB_LOCAL	Local symbols are not visible outside the object file containing their definition. Local symbols of the same name may exist in multiple files without interfering with each other.
STB_GLOBAL	Global symbols are visible to all object files being combined. One file's definition of a global symbol will satisfy another file's undefined reference to the same global symbol.
STB_WEAK	Weak symbols resemble global symbols, but their definitions have lower precedence.
STB_LOPROC through STB_HIPROC	Values in this inclusive range are reserved for processor-specific semantics.

Global and weak symbols differ in two major ways, as described in “Multiply defined symbols” in the *Programming Tools Guide*.

- When the link editor combines several relocatable object files, it does not allow multiple definitions of **STB_GLOBAL** symbols with the same name. On the other hand, if a defined global symbol exists, the appearance of a weak symbol with the same name will not cause an error. The link editor honors the global definition and ignores the weak ones. Similarly, if a common symbol exists (for example, a symbol whose `st_shndx` field holds **SHN_COMMON**), the appearance of a weak symbol with the same name will not cause an error. The link editor honors the common definition and ignores the weak one.

- When the link editor searches archive libraries, it extracts archive members that contain definitions of undefined global symbols. The member's definition may be either a global or a weak symbol. The link editor does not extract archive members to resolve undefined weak symbols. Unresolved weak symbols have a zero value.

In each symbol table, all symbols with **STB_LOCAL** binding precede the weak and global symbols. As described in “Section header” (page 177), a symbol table section's **sh_info** section header member holds the symbol table index for the first non-local symbol.

A symbol's type provides a general classification for the associated entity.

Table 8-11 Symbol types, ELF32_ST_TYPE

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_LOPROC	13
STT_HIPROC	15

STT_NOTYPE	The symbol's type is not specified.
STT_OBJECT	The symbol is associated with a data object, such as a variable, an array, and so forth.
STT_FUNC	The symbol is associated with a function or other executable code.
STT_SECTION	The symbol is associated with a section. Symbol table entries of this type exist primarily for relocation and normally have STB_LOCAL binding.
STT_FILE	Conventionally, the symbol's name gives the name of the source file associated with the object file. A file symbol has STB_LOCAL binding, its section index is SHN_ABS , and it precedes the other STB_LOCAL symbols for the file, if it is present.
STT_LOPROC through STT_HIPROC	Values in this inclusive range are reserved for processor-specific semantics.

Function symbols (those with type `STT_FUNC`) in shared object files have special significance. When another object file references a function from a shared object, the link editor automatically creates a procedure linkage table entry for the referenced symbol. Shared object symbols with types other than `STT_FUNC` will not be referenced automatically through the procedure linkage table.

If a symbol's value refers to a specific location within a section, its section index member, `st_shndx`, holds an index into the section header table. As the section moves during relocation, the symbol's value changes as well, and references to the symbol continue to "point" to the same location in the program. Some special section index values give other semantics.

- SHN_ABS** The symbol has an absolute value that will not change because of relocation.
- SHN_COMMON** The symbol labels a common block that has not yet been allocated. The symbol's value gives alignment constraints, similar to a section's `sh_addralign` member. The link editor will allocate the storage for the symbol at an address that is a multiple of `st_value`. The symbol's size tells how many bytes are required.
- SHN_UNDEF** This section table index means the symbol is undefined. When the link editor combines this object file with another that defines the indicated symbol, this file's references to the symbol will be linked to the actual definition.

The symbol table entry for index 0 (`STN_UNDEF`) is reserved and holds the following values:

Table 8-12 Symbol table entry: index 0

Name	Value	Note
<code>st_name</code>	0	No name
<code>st_value</code>	0	Zero value
<code>st_size</code>	0	No size
<code>st_info</code>	0	No type, local binding
<code>st_other</code>	0	
<code>st_shndx</code>	<code>SHN_UNDEF</code>	No section

Symbol values

Symbol table entries for different object file types have slightly different interpretations for the `st_value` member.

- In relocatable files, `st_value` holds alignment constraints for a symbol whose section index is `SHN_COMMON`.
- In relocatable files, `st_value` holds a section offset for a defined symbol. `st_value` is an offset from the beginning of the section that `st_shndx` identifies.
- In executable and shared object files, `st_value` holds a virtual address. To make these files' symbols more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation) for which the section number is irrelevant.

Although the symbol table values have similar meanings for different object files, the data allows efficient access by the appropriate programs.

Relocation

Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. Relocatable files must have *relocation entries* which are necessary because they contain information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image.

```
typedef struct {
    Elf32_Addr    r_offset;
    Elf32_Word    r_info;
} Elf32_Rel;
```

```
typedef struct {
    Elf32_Addr    r_offset;
    Elf32_Word    r_info;
    Elf32_Sword   r_addend;
} Elf32_Rela;
```

`r_offset` This member gives the location at which to apply the relocation action. For a relocatable file, the value is the byte offset from the beginning of the section to the storage unit affected by the relocation. For an executable file or a shared object, the value is the virtual address of the storage unit affected by the relocation.

`r_info` This member gives both the symbol table index with respect to which the relocation must be made, and the type of relocation to apply. For example, a call instruction's relocation

entry would hold the symbol table index of the function being called. If the index is **STN_UNDEF**, the undefined symbol index, the relocation uses 0 as the *symbol value*. Relocation types are processor-specific; descriptions of their behavior appear below. When the text below refers to a relocation entry's relocation type or symbol table index, it means the result of applying **ELF32_R_TYPE** or **ELF32_R_SYM**, respectively, to the entry's `r_info` member.

```
#define ELF32_R_SYM(i)      ((i)>>8)
#define ELF32_R_TYPE(i)    ((unsigned char)(i))
#define ELF32_R_INFO(s,t)  (((s)<<8)+(unsigned char)(t))
```

`r_addend` This member specifies a constant addend used to compute the value to be stored into the relocatable field.

As specified previously, only **Elf32_Rela** entries contain an explicit addend. Entries of type **Elf32_Rel** store an implicit addend in the location to be modified. Depending on the processor architecture, one form or the other might be necessary or more convenient. Consequently, an implementation for a particular machine may use one form exclusively or either form depending on context.

A relocation section references two other sections: a symbol table and a section to modify. The section header's `sh_info` and `sh_link` members specify these relationships. Relocation entries for different object files have slightly different interpretations for the `r_offset` member.

NOTE See "Section header" (page 177), for more information.

- In relocatable files, `r_offset` holds a section offset. The relocation section itself describes how to modify another section in the file; relocation offsets designate a storage unit within the second section.
- In executable and shared object files, `r_offset` holds a virtual address. To make these files' relocation entries more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation).

Although the interpretation of `r_offset` changes for different object files to allow efficient access by the relevant programs, the relocation types' meanings stay the same.

Program execution

This section describes the object file information and system actions that create running programs. Some information here applies to all systems; information specific to one processor resides in sections marked accordingly.

Executable and shared object files statically represent programs. To execute such programs, the system uses the files to create dynamic program representations, or process images.

Program header

An executable or shared object file's program header table is an array of structures, each describing a segment or other information the system needs to prepare the program for execution. An object file segment contains one or more sections.

NOTE See "Segment contents" (page 198) for more information.

Program headers are meaningful only for executable and shared object files. A file specifies its own program header size with the ELF header's `e_phentsize` and `e_phnum` members. See "ELF header" (page 171) for more information.

```
typedef struct {
    Elf32_Word    p_type;
    Elf32_Off     p_offset;
    Elf32_Addr    p_vaddr;
    Elf32_Addr    p_paddr;
    Elf32_Word    p_filesz;
    Elf32_Word    p_memsz;
    Elf32_Word    p_flags;
    Elf32_Word    p_align;
} Elf32_Phdr;
```

<code>p_type</code>	This member tells what kind of segment this array element describes or how to interpret the array element's information. Type values and their meanings appear in Table 8-4, "Section types, <code>sh_type</code> " (page 180).
<code>p_offset</code>	This member gives the offset from the beginning of the file at which the first byte of the segment resides.
<code>p_vaddr</code>	This member gives the virtual address at which the first byte of the segment resides in memory.

<code>p_paddr</code>	On systems for which physical addressing is relevant, this member is reserved for the segment's physical address. Because UNIX System V ignores physical addressing for application programs, this member has unspecified contents for executable files and shared objects.
<code>p_filesz</code>	This member gives the number of bytes in the file image of the segment; it may be zero.
<code>p_memsz</code>	This member gives the number of bytes in the memory image of the segment; it may be zero.
<code>p_flags</code>	This member gives flags relevant to the segment. Defined flag values appear in Table 8-4, "Section types, <code>sh_type</code> " (page 180).
<code>p_align</code>	Loadable process segments must have congruent values for <code>p_vaddr</code> and <code>p_offset</code> , modulo the page size. See "Program values" for more information. This member gives the value to which the segments are aligned in memory and in the file. Values 0 and 1 mean no alignment is required. Otherwise, <code>p_align</code> should be a positive, integral power of 2, and <code>p_vaddr</code> should equal <code>p_offset</code> , modulo <code>p_align</code> .

Some entries describe process segments; others give supplementary information and do not contribute to the process image. Segment entries may appear in any order, except as explicitly noted in Table 8-13, "Segment types, `p_type`" (this page). Defined type values follow; other values are reserved for future use.

Table 8-13 Segment types, `p_type`

Name	Value
<code>PT_NULL</code>	0
<code>PT_LOAD</code>	1
<code>PT_DYNAMIC</code>	2
<code>PT_INTERP</code>	3
<code>PT_NOTE</code>	4
<code>PT_SHLIB</code>	5
<code>PT_PHDR</code>	6
<code>PT_LOPROC</code>	0x70000000
<code>PT_HIPROC</code>	0x7fffffff

`PT_NULL` The array element is unused; other members' values are undefined. This type lets the program header table have ignored entries.

PT_LOAD	The array element specifies a loadable segment, described by <code>p_filesz</code> and <code>p_memsz</code> . The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size (<code>p_memsz</code>) is larger than the file size (<code>p_filesz</code>), the "extra" bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size may not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the <code>p_vaddr</code> member.
PT_DYNAMIC	The array element specifies dynamic linking information. See "Dynamic section" (page 206) for more information.
PT_INTERP	The array element specifies the location and size of a null-terminated path name to invoke as an interpreter. This segment type is meaningful only for executable files (though it may occur for shared objects); it may not occur more than once in a file. If it is present, it must precede any loadable segment entry. See "Program interpreter" (page 204) for more information.
PT_NOTE	The array element specifies the location and size of auxiliary information. See "Note section" (page 199) for more information.
PT_SHLIB	This segment type is reserved but has unspecified semantics.
PT_PHDR	The array element, if present, specifies the location and size of the program header table itself, both in the file and in the memory image of the program. This segment type may not occur more than once in a file. Moreover, it may occur only if the program header table is part of the memory image of the program. If it is present, it must precede any loadable segment entry. See "Program interpreter" (page 204) for more information.
PT_LOPROC through PT_HIPROC	Values in this inclusive range are reserved for processor-specific semantics.

NOTE Unless specifically required elsewhere, all program header segment types are optional. A file's program header table may contain only those elements relevant to its contents.

Base address

Executable and shared object files have a base address, which is the lowest virtual address associated with the memory image of the program's object file. One use of the base address is to relocate the memory image of the program during dynamic linking.

An executable or shared object file's base address is calculated during execution from three values: the memory load address, the maximum page size, and the lowest virtual address of a program's loadable segment. The virtual addresses in the program headers might not represent the actual virtual addresses of the program's memory image.

NOTE See "Program loading (processor-specific)" (page 201) for more information.

To compute the base address, determine the memory address associated with the lowest `p_vaddr` value for a `PT_LOAD` segment. Next obtain the base address by truncating the memory address to the nearest multiple of the maximum page size. Depending on the type of file being loaded into memory, the memory address might or might not match the `p_vaddr` values.

Segment permissions

A program to be loaded by the system must have at least one loadable segment (although this is not required by the file format). When the system creates loadable segments' memory images, it gives access permissions as specified in the `p_flags` member. All bits included in the `PF_MASKPROC` mask are reserved for processor-specific semantics.

Table 8-14 Segment flag bits, `p_flags`

Name	Value	Meaning
<code>PF_X</code>	<code>0x1</code>	Execute
<code>PF_W</code>	<code>0x2</code>	Write
<code>PF_R</code>	<code>0x4</code>	Read
<code>PF_MASKPROC</code>	<code>0xf0000000</code>	Unspecified

If a permission bit is 0, that type of access is denied. Actual memory permissions depend on the memory management unit, which may vary from one system to another. Although all flag combinations are valid, the system may grant more access than requested. However, a segment will never have write permission unless it is specified explicitly. The following table shows both the exact flag interpretation and the allowable flag interpretation.

Table 8-15 Segment permissions

Flags	Value	Exact	Allowable
none	0	All access denied	All access denied
PF_X	1	Execute only	Read, execute
PF_W	2	Write only	Read, write, execute
PF_W + PF_X	3	Write, execute	Read, write, execute
PF_R	4	Read only	Read, execute
PF_R + PF_X	5	Read, execute	Read, execute
PF_R + PF_W	6	Read, write	Read, write, execute
PF_R + PF_W + PF_X	7	Read, write, execute	Read, write, execute

For example, typical text segments have read and execute — but not write — permissions. Data segments normally have read, write, and execute permissions.

Segment contents

An object file segment comprises one or more sections, though this fact is transparent to the program header. Whether the file segment holds one or many sections also is immaterial to program loading. Nonetheless, various data must be present for program execution, dynamic linking, and so on. Figure 8-9, “Text segment” (this page) illustrates segment contents in general terms. The order and membership of sections within a segment may vary; moreover, processor-specific constraints may alter the examples below.

Text segments contain read-only instructions and data, typically including the following sections described earlier in this chapter. Other sections may also reside in loadable segments; these examples are not meant to give complete and exclusive segment contents.

.text
.rodata
.hash
.dynsym
.dynstr
.plt
.rel.got

Figure 8-9 Text segment

Data segments contain writable data and instructions, typically including the following sections.

.data
.dynamic
.got
.bss

Figure 8-10 Data segment

A `PT_DYNAMIC` program header element points at the `.dynamic` section. See “Dynamic section” (page 206) for more information. The `.got` and `.plt` sections also hold information related to position-independent code and dynamic linking. Although the `.plt` appears in a text segment in the previous table, it may reside in a text or a data segment, depending on the processor.

As described in “Section header” (page 177) in the first part of this chapter, the `.bss` section has the type `SHT_NOBITS`. Although it occupies no space in the file, it contributes to the segment’s memory image. Normally, these uninitialized data reside at the end of the segment, thereby making `p_memsz` larger than `p_filesz` in the associated program header element.

Note section

Sometimes a vendor or system builder needs to mark an object file with special information that other programs will check for conformance, compatibility, and so forth. Sections of type `SHT_NOTE` and program header elements of type `PT_NOTE` can be used for this purpose. The note information in sections and program header elements holds a variable amount of entries, each of which is an array of 4-byte words in the format of the target processor. Labels appear in Figure 8-11, “Note information” (this page) to help explain note information organization, but they are not part of the specification.

namesz
descsz
type
name ...
desc ...

Figure 8-11 Note information

<code>namesz</code> and <code>name</code>	The first <code>namesz</code> bytes in <code>name</code> contain a null-terminated character representation of the entry's owner or originator. There is no formal mechanism for avoiding name conflicts. By convention, vendors use their own name, such as XYZ Computer Company, as the identifier. If no name is present, <code>namesz</code> contains 0. Padding is present, if necessary, to ensure 4-byte alignment for the descriptor. Such padding is not included in <code>namesz</code> .
<code>descsz</code> and <code>desc</code>	The first <code>descsz</code> bytes in <code>desc</code> hold the note descriptor. If no descriptor is present, <code>descsz</code> contains 0. Padding is present, if necessary, to ensure 4-byte alignment for the next note entry. Such padding is not included in <code>descsz</code> .
<code>type</code>	This word gives the interpretation of the descriptor. Each originator controls its own types; multiple interpretations of a single type value may exist. Thus, a program must recognize both the name and the type to recognize a descriptor. Types currently must be non-negative.

To illustrate, the following note segment holds two entries.

	+0	+1	+2	+3	
namesz	7				No descriptor
descsz	0				
type	1				
name	X	Y	Z		
	C	o	\0	pad	
namesz	7				
descsz	8				
type	3				
name	X	Y	Z		
	C	o	\0	pad	
desc	word 0				
	word 1				

Figure 8-12 Example note segment

NOTE The system reserves note information with no name (`namesz==0`) and with a zero-length name (`name[0]=='\0'`) but currently defines no types. All other names must have at least one non-null character.

Program loading (processor-specific)

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When — and if — the system physically reads the file depends on the program's execution behavior, system load, and so forth. A process does not require a physical page unless it references the logical page during execution, and processes commonly leave many pages unreferenced. Therefore delaying physical reads frequently obviates them, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size. By aligning segments to the maximum page size, the files will be suitable for paging regardless of physical page size.

The following examples show 8 K alignment; virtual addresses and file offsets for segments are congruent modulo 8 K (0x2000). Actual minimum page sizes are machine specific.

File offset	File	Virtual address
0	ELF header	
	Program header table	
	Other information	
0x100	Text segment	0x80000100
	...	
	0x2be00 bytes	0x8002beff
0x2bf00	Data segment	0x8002df00
	...	
	0x4e00 bytes	0x80032cff
0x30d00	Other information	
	...	

Figure 8-13 Executable file

Table 8-16 Program header segments

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x80000100	0x8002df00
p_paddr	unspecified	unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R + PF_X	PF_R + PF_W + PF_X
p_align	0x2000	0x2000

Although the example's file offsets and virtual addresses are congruent modulo the maximum page size for both text and data, up to four file pages hold impure text or data (depending on page size and file system block size).

- The first text page contains the ELF header, the program header table, and other information.
- The last text page holds a copy of the beginning of data.
- The first data page has a copy of the end of text.
- The last data page may contain file information not relevant to the running process.

Logically, the system enforces the memory permissions as if each segment were complete and separate; segments' addresses are adjusted to ensure each logical page in the address space has a single set of permissions. In the previous example, the region of the file holding the end of text and the beginning of data will be mapped twice: at one virtual address for text and at a different virtual address for data.

The end of the data segment requires special handling for uninitialized data, which the system defines to begin with zero values. Thus if a file's last data page includes information not in the logical memory page, the extraneous data must be set to zero, not the unknown contents of the executable file. *Impurities* in the other three pages are not logically part of the process image; whether the system expunges them is unspecified. The memory image for this program follows, assuming 2KB (0x800) pages. For simplicity, this example illustrates only one page size.

Virtual address	Contents	Segment
0x80000000	Header padding 0x100 bytes	Text
0x80000100	Text segment	
	...	
	0x2be00 bytes	
0x8002bf00	Data padding 0x100 bytes	Data
0x8002d800	Text padding 0x700 bytes	
0x8002df00	Data segment	
	...	
	0x4e00 bytes	
0x80032d00	Uninitialized data 0x1024 zero bytes	
0x80033d24	Page padding 0x2dc zero bytes	

Figure 8-14 Process image segments

One aspect of segment loading differs between executable files and shared objects. Executable file segments typically contain absolute code. To let the process execute correctly, the segments must reside at the virtual addresses used to build the executable file. Thus the system uses the `p_vaddr` values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code.

NOTE See “Link editing” in the *Programming Tools Guide* for more information.

This lets a segment’s virtual address change from one process to another, without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the segments’ relative positions. Because position-independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file. Table 8-17, “Example shared object segment addresses” (page 204) shows possible shared object

virtual address assignments for several processes, illustrating constant relative positioning. The table also illustrates the base address computations. The maximum page size used in Table 8-17, “Example shared object segment addresses” (this page) is 8KB.

Table 8-17 Example shared object segment addresses

Source	Text	Data	Base address
File	0x200	0x2a400	0x0
Process 1	0xc0080200	0xc00aa400	0xc0080000
Process 2	0xc0082200	0xc00ac400	0xc0082000
Process 3	0xd00c0200	0xd00ea400	0xd00c0000
Process 4	0xd00c6200	0xd00f0400	0xd00c6000

Program interpreter

An executable file may have one `PT_INTERP` program header element. During `exec()`, the system retrieves a path name from the `PT_INTERP` segment and creates the initial process image from the interpreter file’s segments. Instead of using the original executable file’s segment images, the system composes a memory image for the interpreter. It then is the interpreter’s responsibility to receive control from the system and provide an environment for the application program.

The interpreter receives control in one of two ways. First, it may receive a file descriptor to read the executable file, positioned at the beginning. It can use this file descriptor to read and/or map the executable file’s segments into memory. Second, depending on the executable file format, the system may load the executable file into memory instead of giving the interpreter an open file descriptor. With the possible exception of the file descriptor, the interpreter’s initial process state matches what the executable file would have received. The interpreter itself may not require a second interpreter. An interpreter may be either a shared object or an executable file.

- A shared object (the normal case) is loaded as position-independent, with addresses that may vary from one process to another; the system creates its segments in the dynamic segment area used by `mmap()` and related services. Consequently, a shared object interpreter typically will not conflict with the original executable file’s original segment addresses.
- An executable file is loaded at fixed addresses; the system creates its segments using the virtual addresses from the program header table. Consequently, an executable file interpreter’s virtual addresses may collide with the first executable file; the interpreter is responsible for resolving conflicts.

Dynamic linker

When building an executable file that uses dynamic linking, the link editor adds a program header element of type `PT_INTERP` to an executable file, telling the system to invoke the dynamic linker as the program interpreter. `exec()` and the dynamic linker cooperate to create the process image for the program, which entails the following actions:

NOTE The location(s) of the system provided dynamic linker(s) are processor specific.

- Adding the executable file's memory segments to the process image;
- Adding shared object memory segments to the process image;
- Performing relocations for the executable file and its shared objects;
- Closing the file descriptor that was used to read the executable file, if one was given to the dynamic linker;
- Transferring control to the program, making it look as if the program had received control directly from `exec()`.

The link editor also constructs various data that assist the dynamic linker for executable and shared object files. As shown previously in "Program header" (page 194), this data resides in loadable segments, making them available during execution.

- A `.dynamic` section with type `SHT_DYNAMIC` holds various data. The structure residing at the beginning of the section holds the addresses of other dynamic linking information.
- The `.hash` section with type `SHT_HASH` holds a symbol hash table.
- The `.got` and `.plt` sections with type `SHT_PROGBITS` hold two separate tables: the global offset table and the procedure linkage table. Sections below explain how the dynamic linker uses and changes the tables to create memory images for object files.

As explained in "Program loading (processor-specific)" (page 201), shared objects may occupy virtual memory addresses that are different from the addresses recorded in the file's program header table. The dynamic linker relocates the memory image, updating absolute addresses before the application gains control. Although the absolute address values would be correct if the library were loaded at the addresses specified in the program header table, this normally is not the case.

If the process environment contains a variable named with a non-null value the dynamic linker processes all relocations before transferring control to the program.

NOTE See “Checking for run-time compatibility” in the *Programming Tools Guide* for more information.

For example, all the following environment entries would specify this behavior.

- LD_BIND_NOW=1
- LD_BIND_NOW=on
- LD_BIND_NOW=off

Otherwise, **LD_BIND_NOW** either does not occur in the environment or has a null value. The dynamic linker is permitted to evaluate procedure linkage table entries slowly, thus avoiding symbol resolution and relocation overhead for functions that are not called.

Dynamic section

If an object file participates in dynamic linking, its program header table will have an element of type **PT_DYNAMIC**. This *segment* contains the `.dynamic` section. A special symbol, `_DYNAMIC`, labels the section, which contains an array of the following structures.

```
typedef struct {
    Elf32_Sword    d_tag;
    union {
        Elf32_Word    d_val;
        Elf32_Addr    d_ptr;
    } d_un;
} Elf32_Dyn;

extern Elf32_Dyn    _DYNAMIC[];
```

For each object with this type, `d_tag` controls the interpretation of `d_un`.

<code>d_val</code>	These Elf32_Word objects represent integer values with various interpretations.
<code>d_ptr</code>	These Elf32_Addr objects represent program virtual addresses. As mentioned previously, a file’s virtual addresses might not match the memory virtual addresses during execution. When interpreting addresses contained in the dynamic structure, the dynamic linker computes actual addresses, based on the original file value and the memory base address. For consistency, files do not contain relocation entries to <i>correct</i> addresses in the dynamic structure.

Table 8-18, “Dynamic array tags, d_tag” (this page) summarizes the tag requirements for executable and shared object files. If a tag is marked *mandatory*, the dynamic linking array must have an entry of that type. Likewise, *optional* means an entry for the tag may appear but is not required.

Table 8-18 Dynamic array tags, d_tag

Name	Value	d_un	Executable	Shared object
DT_NULL	0	ignored	mandatory	mandatory
DT_NEEDED	1	d_val	optional	optional
DT_PLTRELSZ	2	d_val	optional	optional
DT_PLTGOT	3	d_ptr	optional	optional
DT_HASH	4	d_ptr	mandatory	mandatory
DT_STRTAB	5	d_ptr	mandatory	mandatory
DT_SYMTAB	6	d_ptr	mandatory	mandatory
DT_RELA	7	d_ptr	mandatory	optional
DT_RELASZ	8	d_val	mandatory	optional
DT_RELAENT	9	d_val	mandatory	optional
DT_STRSZ	10	d_val	mandatory	mandatory
DT_SYMENT	11	d_val	mandatory	mandatory
DT_INIT	12	d_ptr	optional	optional
DT_FINI	13	d_ptr	optional	optional
DT_SONAME	14	d_val	ignored	optional
DT_RPATH	15	d_val	optional	ignored
DT_SYMBOLIC	16	ignored	ignored	optional
DT_REL	17	d_ptr	mandatory	optional
DT_RELSZ	18	d_val	mandatory	optional
DT_RELENT	19	d_val	mandatory	optional
DT_PLTREL	20	d_val	optional	optional
DT_DEBUG	21	d_ptr	optional	ignored
DT_TEXTREL	22	ignored	optional	optional
DT_JMPREL	23	d_ptr	optional	optional
DT_LOPROC	0x70000000	unspecified	unspecified	unspecified
DT_HIPROC	0x7fffffff	unspecified	unspecified	unspecified

DT_NULL An entry with a **DT_NULL** tag marks the end of the **_DYNAMIC** array.

DT_NEEDED This element holds the string table offset of a null-terminated string, giving the name of a needed library. The offset is an index into the table recorded in the **DT_STRTAB** entry. See “Shared object dependencies” (page 210) for more information about these names. The dynamic array may contain multiple entries with this type. These entries’ relative order is significant, though their relation to entries of other types is not.

DT_PLTRELSZ	This element holds the total size, in bytes, of the relocation entries associated with the procedure linkage table. If an entry of type DT_JMPREL is present, a DT_PLTRELSZ must accompany it.
DT_PLTGOT	This element holds an address associated with the procedure linkage table and/or the global offset table. Semantics are processor specific.
DT_HASH	This element holds the address of the symbol hash table, described in “Hash table” (page 212).
DT_STRTAB	This element holds the address of the string table, described in the first part of this chapter. Symbol names, library names, and other strings reside in this table.
DT_SYMTAB	This element holds the address of the symbol table, described in the first part of this chapter, with Elf32_Sym entries for the 32-bit class of files.
DT_RELA	This element holds the address of a relocation table, described in the first part of this chapter. Entries in the table have explicit addends, such as Elf32_Rela for the 32-bit file class. An object file may have multiple relocation sections. When building the relocation table for an executable or shared object file, the link editor catenates those sections to form a single table. Although the sections remain independent in the object file, the dynamic linker sees a single table. When the dynamic linker creates the process image for an executable file or adds a shared object to the process image, it reads the relocation table and performs the associated actions. If this element is present, the dynamic structure must also have DT_RELASZ and DT_RELAENT elements. When relocation is “mandatory” for a file, either DT_RELA or DT_REL may occur (both are permitted but not required).
DT_RELASZ	This element holds the total size, in bytes, of the DT_RELA relocation table.
DT_RELAENT	This element holds the size, in bytes, of the DT_RELA relocation entry.
DT_STRSZ	This element holds the size, in bytes, of the string table.
DT_SYMENT	This element holds the size, in bytes, of a symbol table entry.
DT_INIT	This element holds the address of the initialization function. See “Initialization and termination functions” (page 213) for more information.

DT_FINI	This element holds the address of the termination function. See “Initialization and termination functions” (page 213) for more information.
DT_SONAME	This element holds the string table offset of a null-terminated string, giving the name of the shared object. The offset is an index into the table recorded in the DT_STRTAB entry. NOTE See “Shared object dependencies” (page 210) for more information about these names.
DT_RPATH	This element holds the string table offset of a null-terminated search library search path string. The offset is an index into the table recorded in the DT_STRTAB entry.
DT_SYMBOLIC	This element’s presence in a shared object library alters the dynamic linker’s symbol resolution algorithm for references within the library. Instead of starting a symbol search with the executable file, the dynamic linker starts from the shared object itself. If the shared object fails to supply the referenced symbol, the dynamic linker then searches the executable file and other shared objects as usual.
DT_REL	This element is similar to DT_RELA , except its table has implicit addends, such as Elf32_Rel for the 32-bit file class. If this element is present, the dynamic structure must also have DT_RELSZ and DT_RELENT elements.
DT_RELSZ	This element holds the total size, in bytes, of the DT_REL relocation table.
DT_RELENT	This element holds the size, in bytes, of the DT_REL relocation entry.
DT_PLTREL	This member specifies the type of relocation entry to which the procedure linkage table refers. The d_val member holds DT_REL or DT_RELA , as appropriate. All relocations in a procedure linkage table must use the same relocation.
DT_DEBUG	This member is used for debugging.
DT_TEXTREL	This member’s absence signifies that no relocation entry should cause a modification to a non-writable segment, as specified by the segment permissions in the program header table. If this member is present, one or more relocation entries might request modifications to a non-writable segment, and the dynamic linker can prepare accordingly.

DT_JMPREL If present, this entry's `d_ptr` member holds the address of relocation entries associated solely with the procedure linkage table. Separating these relocation entries lets the dynamic linker ignore them during process initialization, if lazy binding is enabled. If this entry is present, the related entries of types **DT_PLTRELSZ** and **DT_PLTREL** must also be present.

DT_LOPROC through **DT_HIPROC**

Values in this inclusive range are reserved for processor-specific semantics.

Except for the **DT_NULL** element at the end of the array, and the relative order of **DT_NEEDED** elements, entries may appear in any order. Tag values not appearing in the table are reserved.

Shared object dependencies

When the link editor processes an archive library, it extracts library members and copies them into the output object file. These statically linked services are available during execution without involving the dynamic linker. Shared objects also provide services, and the dynamic linker must attach the proper shared object files to the process image for execution. Thus executable and shared object files describe their specific dependencies. For further information, see “Link editing” in the *Programming Tools Guide*.

When the dynamic linker creates the memory segments for an object file, the dependencies (recorded in **DT_NEEDED** entries of the dynamic structure) tell what shared objects are needed to supply the program's services. By repeatedly connecting referenced shared objects and their dependencies, the dynamic linker builds a complete process image. When resolving symbolic references, the dynamic linker examines the symbol tables with a breadth-first search. It first looks at the symbol table of the executable program itself, then at the symbol tables of the **DT_NEEDED** entries (in order), and then at the second level **DT_NEEDED** entries.

NOTE Even when a shared object is referenced multiple times in the dependency list, the dynamic linker will connect the object only once to the process.

Names in the dependency list are copies either of the **DT_SONAME** strings or the path names of the shared objects used to build the object file. For example, if the link editor builds an executable file using one shared object with a **DT_SONAME** entry of `lib1` and another shared object library with the path name `/usr/lib/lib2`, the executable file will contain `lib1` and `/usr/lib/lib2` in its dependency list.

If a shared object name has one or more slash (/) characters anywhere in the name, such as `/usr/lib/lib2` or `directory/file`, the dynamic linker uses that string directly as the path name. If the name has no slashes, such as `lib1`, three facilities specify shared object path searching, with the following caveats:

1. The dynamic array tag **DT_RPATH** may give a string that holds a list of directories, separated by colons (:). For example, the string `/home/dir/usr/lib:/home/dir2/usr/lib:` tells the dynamic linker to search first the directory `/home/dir/lib`, then `/home/dir2/usr/lib`, and then the current directory to find dependencies.
2. A variable called **LD_LIBRARY_PATH** in the process environment may hold a list of directories as above, optionally followed by a semicolon (;) and another directory list. The following values would be equivalent to the previous example:
 - **LD_LIBRARY_PATH**=`/home/dir/usr/lib:/home/dir2/usr/lib:`
 - **LD_LIBRARY_PATH**=`/home/dir/usr/lib;/home/dir2/usr/lib:`
 - **LD_LIBRARY_PATH**=`/home/dir/usr/lib:/home/dir2/usr/lib:`

All **LD_LIBRARY_PATH** directories are searched after those from **DT_RPATH**. Although some programs (such as the link editor) treat the lists before and after the semicolon differently, the dynamic linker does not. Nevertheless, the dynamic linker accepts the semicolon notation, with the semantics described previously.

3. If the other two groups of directories fail to locate the desired library, the dynamic linker searches `/usr/lib`.

NOTE For security, the dynamic linker ignores environmental search specifications (such as **LD_LIBRARY_PATH**) for set-user and set-group ID programs. It does, however, search **DT_RPATH** directories and `/usr/lib`.

Hash table

A hash table of `Elf32_Word` objects supports symbol table access. Labels appear in Figure 8-16, “Symbol hash table” (this page) to help explain the hash table organization, but they are not part of the specification.

<code>nbucket</code>
<code>nchain</code>
<code>bucket[0]</code>
...
<code>bucket[nbucket - 1]</code>
<code>chain[0]</code>
...
<code>chain[nchain - 1]</code>

Figure 8-16 Symbol hash table

The bucket array contains `nbucket` entries, and the chain array contains `nchain` entries; indexes start at 0. Both bucket and chain hold symbol table indexes. Chain table entries parallel the symbol table. The number of symbol table entries should equal `nchain`; so symbol table indexes also select chain table entries. A hashing function accepts a symbol name and returns a value that may be used to compute a bucket index. Consequently, if the hashing function returns the value “x” for some name, **bucket[x*nbucket]** gives an index, “y”, into both the symbol table and the chain table. If the symbol table entry is not the one desired, `chain[y]` gives the next symbol table entry with the same hash value. One can follow the chain links until either the selected symbol table entry holds the desired name or the chain entry contains the value `STN_UNDEF`.

```
unsigned long elf_hash(const unsigned char *name) {
    unsigned long    h = 0, g;

    while (*name)
    {
        h = (h << 4) + *name++;
        if (g = h & 0xf0000000)
            h ^= g >> 24;
        h &= ~g;
    }
    return h;
}
```

Initialization and termination functions

After the dynamic linker has built the process image and performed the relocations, each shared object gets the opportunity to execute some initialization code. These initialization functions are called in no specified order, but all shared object initializations happen before the executable file gains control.

Similarly, shared objects may have termination functions, which are executed with the **atexit(S)** mechanism after the base process begins its termination sequence. See **atexit(S)** for more information. Once again, the order in which the dynamic linker calls termination functions is unspecified.

Shared objects designate their initialization and termination functions through the **DT_INIT** and **DT_FINI** entries in the dynamic structure. See “Dynamic section” (page 206) for more information. Typically, the code for these functions resides in the `.init` and `.fini` sections. See “Section header” (page 177) for more information.

NOTE Although the **atexit()** termination processing normally will be done, it is not guaranteed to have executed upon process death. In particular, the process will not execute the termination processing if it calls **_exit()** or if the process dies because it received a signal that it neither caught nor ignored.

Chapter 9

Enhanced *asm* facility

Although the ability to write portable code is one reason for using the C language, sometimes it is necessary to introduce machine-specific assembly language instructions into C code. This need arises most often within operating system code that must deal with hardware registers that would otherwise be inaccessible from C. The **asm** facility makes it possible to introduce this assembly code.

In earlier versions of C the **asm** facility was primitive. You included a line that looked like a call on the function **asm**, which took one argument, a string:

```
asm("assembly instruction here");
```

Unfortunately this technique has shortcomings when the assembly instruction needs to reference C language operands. You have to guess the register or stack location into which the compiler would put the operand and encode that location into the instruction. If the compiler's allocation scheme changed, or, more likely, if the C code surrounding the **asm** changed, the correct location for the operand in the **asm** would also change. You would have to be aware that the C code would affect the **asm** and change it accordingly.

The new facility presented here is upwardly compatible with old code, since it retains the old capability. In addition, it allows you to define **asm** macros that describe how machine instructions should be generated when their operands take particular forms that the compiler recognizes, such as register or stack variables.

NOTE Although this enhanced **asm** facility is easier to use than before, you are still strongly discouraged from using it for routine applications because those applications will not be portable to different machines. The primary intended use of the **asm** facility is to help implement operating systems in a clean way.

The optimizer (`cc -O`) may work incorrectly on C programs that use the **asm** facility, particularly when the **asm** macros contain instructions or labels that are unlike those that the C compiler generates. Furthermore, you may need to rewrite **asm** code in the future to maximize its benefits as new optimization technology is introduced into the compilation system.

Definition of terms

The following terms are used:

asm macro	An asm macro is the mechanism by which programs use the enhanced asm facility. asm macros have a definition and uses . The definition includes a set of pattern/body pairs. Each pattern describes the storage modes that the actual arguments must match for the asm macro body to be expanded. The uses resemble C function calls.
storage mode	The storage mode , or mode , of an asm macro argument is the compiler's idea of where the argument can be found at run time. Examples are "in a register" or "in memory".
pattern	A pattern specifies the modes for each of the arguments of an asm macro. When the modes in the pattern all match those of the use, the corresponding body is expanded.
asm macro body	The asm macro body , or body , is the portion of code that will be expanded by the compiler when the corresponding pattern matches. The body may contain references to the formal parameters, in which case the compiler substitutes the corresponding assembly language code.

Example

The following example uses a machine with an **spl** instruction for setting machine interrupt priority levels. **spl** takes one operand, which must be in a register. Nevertheless, it would be convenient to have a function that produces in-line code to set priority levels, uses the **spl** instruction, and works with register variables or constants.

This example consists of two parts, the definition of the **asm** macro, and its use.

Definition

Define an **asm** macro, called **SPL**:

```
asm void SPL(newpri)
{
%      reg newpri;
      spl newpri
%      con newpri;
      movw newpri,%r0
      spl %r0
}
```

The lines that begin with “%” are patterns. If the arguments at the time the macro is called match the **storage modes** in a pattern, the code that follows the pattern line will be expanded.

Use

The table below shows the (assembly) code that the compiler would generate with two different uses of **SPL**. It uses the following introductory code (along with the above definition):

```
f() {
    register int i;
```

code...	matches...	generates...
SPL(i);	% reg	spl %r8
SPL(3);	% con	movw &3,%r0 spl %r0

The first use of **SPL** has a register variable as its argument (assuming that *i* actually gets allocated to a register). This argument has a storage mode that matches **reg**, the storage mode in the first pattern. Therefore the compiler expands the first code body. Note that **newpri**, the formal parameter in the definition, has been replaced in the expanded code by the compiler’s idea of the assembly time name for the variable *i*, namely **%r8**. Similarly, the second use of **SPL** has a constant as its argument, which leads to the compiler’s choosing the second pattern. Here again **newpri** has been replaced by the assembly time form for the constant, **&3**.

Using asm macros

The enhanced **asm** facility allows you to define constructs that behave syntactically like static C functions. Each **asm** macro has one definition and zero or more uses per source file. The definition must appear in the same file with the uses (or be **#included**), and the same **asm** macro may be defined multiply (and differently) in several files.

The **asm** macro definition declares a return type for the macro code, specifies patterns for the formal parameters, and provides bodies of code to expand when the patterns match. When it encounters an **asm** macro call, the compiler replaces uses of the formal parameters by its idea of the assembly language locations of the actual arguments as it expands the code body. This constitutes an important difference between C functions and **asm** macros. An **asm** macro can therefore have the effect of changing the value of its arguments, whereas a C function can only change a copy of its argument values.

The uses of an **asm** macro look exactly like normal C function calls. They may be used in expressions and they may return values. The arguments to an **asm** macro may be arbitrary expressions, except that they may not contain uses of the same or other **asm** macros.

When the argument to an **asm** macro is a function name or structure, the compiler generates code to compute a pointer to the structure or function, and the resulting pointer is used as the actual argument of the macro.

Definition

The syntactic descriptions that follow are presented in the style used in Chapter 3, “C language compiler” in the *Programming Tools Guide*. The syntactic classes **type-specifier**, **identifier**, and **parameter-list** have the same form as in that chapter. A syntactic description enclosed in square brackets (“[]”) is optional, unless the right bracket is followed by a “+”. A “+” means “one or more repetitions” of a description. Similarly, “*” means “zero or more repetitions”.

```
asm macro:
    asm [ type-specifier ] identifier ( [ parameter-list ] )
    {
        [ storage-mode-specification-line
          asm-body ] *
    }
```

An **asm** macro consists of the keyword **asm**, followed by what looks like a C function declaration. Inside the macro body there are one or more pairs of **storage-mode-specification-line(s)** (patterns) and corresponding **asm-body(ies)**. If the **type-specifier** is other than **void**, the **asm** macro should return a value of the declared type.

```
storage-mode-specification-line:
    % [ storage-mode [ identifier [ , identifier ]* ] ; ]+
```

A **storage-mode-specification-line** consists of a single line (no continuation with \ is permitted) that begins with % and contains the names, (**identifier**)(s) and **storage mode(s)** of the formal parameters. Modes for all formal parameters must be given in each **storage-mode-specification-line** (except for **error**). The % must be the first character on a line. If an **asm** macro has no **parameter-list**, the **storage-mode-specification-line** may be omitted.

Storage modes

These are the storage modes that the compiler recognizes in **asm** macros:

treg	Compiler-selected temporary register.
ureg	C register variable that the compiler has allocated in a machine register.
reg	treg or ureg .
con	Compile time constant.
mem	A mem operand matches any allowed machine addressing mode, including reg and con .
lab	Compiler-generated unique label. The identifier(s) that are specified as being of mode lab do not appear as formal parameters in the asm macro definition, unlike the preceding modes. Such identifiers must be unique.
error	Generate a compiler error. This mode exists to allow you to flag errors at compile time if no appropriate pattern exists for a set of actual arguments.

asm body

The **asm** body represents (presumed) assembly code that the compiler will generate when the modes for all of the formal parameters match the associated pattern. Syntactically, the **asm** body consists of the text between two pattern lines (that begin with %) or between the last pattern line and the } that ends the **asm** macro. C language comment lines are not recognized as such in the **asm** body. Instead they are simply considered part of the text to be expanded.

Formal parameter names may appear in any context in the **asm** body, delimited by non-alphanumeric characters. For each instance of a formal parameter in the **asm** body the compiler substitutes the appropriate assembly language operand syntax that will access the actual argument at run time. As an example, if one of the actual arguments to an **asm** macro is *x*, an automatic variable, a string like **4(%fp)** would be substituted for occurrences of the corresponding formal parameter. An important consequence of this macro substitution behavior is that **asm** macros can change the value of their arguments. Note that this is different from standard C semantics.

For **lab** parameters a unique label is chosen for each new expansion.

If an **asm** macro is declared to return a value, it must be coded to return a value of the proper type in the machine register that is appropriate for the implementation.

An implementation restriction requires that no line in the **asm** body may start with %.

Writing asm macros

Here are some guidelines for writing **asm** macros.

1. Know the implementation. You must be familiar with the C compiler and assembly language with which you are working. You can consult the *Application Binary Interface* for your machine for the details of function calling and register usage conventions.
2. Observe register conventions. You should be aware of which registers the C compiler normally uses for scratch registers or register variables. An **asm** macro may alter scratch registers at will, but the values in register variables must be preserved. You must know in which register(s) the compiler returns function results.
3. Handle return values. **asm** macros may “return” values. That means they behave as if they were actually functions that had been called via the usual function call mechanism. **asm** macros must therefore mimic C’s behavior in that respect, passing return values in the same place as normal C functions. Note that **float** and **double** results sometimes get returned in different registers from integer-type results. On some machine architectures, C functions return pointers in different registers from those used for scalars. Finally, **structs** may be returned in a variety of implementation-dependent ways.
4. Cover all cases. The **asm** macro patterns should cover all combinations of storage modes of the parameters. The compiler attempts to match patterns in the order of their appearance in the **asm** macro definition.

There are two escape mechanisms for the matching process. If the compiler encounters a storage mode of **error** while attempting to find a matching pattern, it generates a compile time error for that particular **asm** macro call. If the **asm** macro definition lacks an **error** storage mode and no pattern matches, the compiler generates a normal function call for a function having the same name as the **asm** macro. Note that such a function would have to be defined in a different source file, since its name would conflict with that of the **asm** macro.

5. Beware of argument handling. **asm** macro arguments are used for macro substitution. Thus, unlike normal C functions, **asm** macros can alter the underlying values that their arguments refer to. Altering argument values is discouraged, however, because doing so would make it impossible to substitute an equivalent C function call for the **asm** macro call.
6. **asm** macros are inherently nonportable and implementation-dependent. Although they make it easier to introduce assembly code reliably into C code, the process cannot be made foolproof. You will always need to verify correct behavior by inspection and testing.
7. Debuggers will generally have difficulty with **asm** macros. It may be impossible to set breakpoints within the in-line code that the compiler generates.
8. Because optimizers are highly tuned to the normal code generation sequences of the compiler, using **asm** macros may cause optimizers to produce incorrect code. Generally speaking, any **asm** macro that can be directly replaced by a comparable C function may be optimized safely. However, the sensitivity of an optimizer to **asm** macros varies among implementations and may change with new software releases.

Chapter 10

Mapfile option

The ELF linker, `ld(CP)`, automatically and intelligently maps input sections from object files (*.o* files) to output segments in executable files (*a.out* files). The `mapfile` option to the `ld` command allows you to change the default mapping provided by the ELF linker.

In particular, the `mapfile` option allows you to:

- Declare segments and specify values for segment attributes such as segment type, permissions, addresses, length, and alignment.
- Control mapping of input sections to segments by specifying the attribute values necessary in a section to map to a specific segment (the attributes are section name, section type, and permissions) and by specifying which object file(s) the input sections should be taken from, if necessary.
- Declare a global-absolute symbol that is assigned a value equal to the size of a specified segment (by the linker) and that can be referenced from object files.

NOTE The major purpose of the `mapfile` option is to allow users of *ifiles* (an option previously available to `ld` that used link editor command language directives) to convert to mapfiles. All other facilities previously available for *ifiles*, other than those mentioned above, are not available with the `mapfile` option.

CAUTION When using the `mapfile` option, be aware that you can easily create *a.out* files that do not execute. Therefore, the use of the `mapfile` option is strongly discouraged. `ld` can produce a correct *a.out* without the use of the `mapfile` option. The `mapfile` option is intended for system programming use, not application programming use.

This chapter describes the structure and syntax of a mapfile and the use of the **-M** option to the **ld** command.

Using the mapfile option

To use the mapfile option, you must:

1. Enter mapfile directives into a file (this is your “mapfile”).
2. Enter the following option on the **ld** command line:

-M mapfile

mapfile is the file name of the file you produced in step 1. If the *mapfile* is not in your current directory, you must include the full path name; no default search path exists. (See the **ld**(CP) manual page for information on operation of the **ld** command.)

CAUTION The mapfile option can only be used in static mode. The **-dn** option must accompany the **-M** option on the **ld** command line or **ld** returns a fatal error.

Mapfile structure and syntax

You can enter three types of directives into a mapfile:

- segment declarations
- mapping directives
- size-symbol declarations

Each directive can span more than one line and can have any amount of white space (including new-lines) as long as it is followed by a semicolon. You can enter 0 (zero) or more directives in a mapfile. (Entering zero directives causes **ld** to ignore the mapfile and use its own defaults.) Typically, segment declarations are followed by mapping directives. You would declare a segment and then define the criteria by which a section becomes part of that segment. If you enter a mapping directive or size-symbol declaration without first declaring the segment to which you are mapping (except for built-in segments, explained later), the segment is given default attributes as explained below. This segment is then an “implicitly declared segment”.

Size-symbol declarations can appear anywhere in a mapfile.

The following sections describe each directive type. For all syntax discussions, the following apply:

- All entries in “constant width”, all colons, semicolons, equal signs, and at (@) signs are typed in literally.
- All entries in ***bold-italics*** are substitutables.
- { ... }* means “zero or more”.
- { ... }+ means “one or more”.
- [...] means “optional”.
- *section_names* and *segment_names* follow the same rules as C identifiers where a period (.) is treated as a letter (for example, *.bss* is a valid name).
- *section_names*, *segment_names*, *file_names*, and *symbol_names* are case sensitive; everything else is not case sensitive.
- Spaces (or new-lines) may appear anywhere except before a *number* or in the middle of a name or value.
- Comments beginning with “#” and ending at a new-line may appear anywhere that a space may appear.

Segment declarations

A segment declaration creates a new segment in the *a.out* or changes the attribute values of an existing segment. (An existing segment is one that you previously defined or one of the three built-in segments described below.)

A segment declaration has the following syntax:

segment_name = {*segment_attribute_value*}*;

For each *segment_name*, you can specify any number of *segment_attribute_values* in any order, each separated by a space. (Only one attribute value is allowed for each segment attribute.) The segment attributes and their valid values are as follows:

<i>segment_type</i> :	LOAD NOTE
<i>segment flags</i> :	?[R][W][X]
<i>virtual_address</i> :	V <i>number</i>
<i>physical_address</i> :	P <i>number</i>
<i>length</i> :	L <i>number</i>
<i>alignment</i> :	A <i>number</i>

There are three built-in segments with the following default attribute values:

- text (LOAD, ?RX, *virtual_address*, *physical_address*, *length*, and *alignment* values set to defaults per CPU type)
- data (LOAD, ?RWX, *virtual_address*, *physical_address*, *length*, and *alignment* values set to defaults per CPU type)
- note (NOTE)

ld behaves as if these segments had been declared before your mapfile is read in. See “Mapfile option defaults” (page 231) for more information.

Note the following when entering segment declarations:

- A *number* can be hexadecimal, decimal, or octal, following the same rules as in the C language.
- No space is allowed between the V, P, L, or A and the *number*.
- The *segment_type* value can be either LOAD or NOTE.
- The *segment_type* value defaults to LOAD.
- The *segment_flags* values are R for readable, W for writable, and X for executable. No spaces are allowed between the question mark and the individual flags that make up the *segment_flags* value.
- The *segment_flags* value for a LOAD segment defaults to ?RWX.
- NOTE segments cannot be assigned any segment attribute value other than a *segment_type*.
- Implicitly declared segments default to *segment_type* value LOAD, *segment_flags* value ?RWX, *virtual_address*, *physical_address*, *length*, and *alignment* values set to defaults per CPU type.

NOTE **ld** calculates the addresses and length of the current segment based on the previous segment’s attribute values. Also, even though implicitly declared segments default to “no length limit”, any machine memory limitations still apply.

- LOAD segments can have an explicitly specified *virtual_address* value and/or *physical_address* value, as well as a maximum segment *length* value.
- If a segment has a *segment_flags* value of ? with nothing following, the value defaults to not readable, not writable and not executable.

- The *alignment* value is used in calculating the virtual address of the beginning of the segment. This alignment only affects the segment for which it is specified; other segments still have the default alignment unless their alignments are also changed.
- If any of the *virtual_address*, *physical_address*, or *length* attribute values are not set, *ld* calculates these values as it builds the *a.out*.
- If an *alignment* value is not specified for a segment, it is set to the built-in default. (The default differs from one CPU to another and may even differ between kernel versions. You should check the appropriate documentation for these numbers).
- If both a *virtual_address* and an *alignment* value are specified for a segment, the *virtual_address* value takes priority.
- If a *virtual_address* value is specified for a segment, the alignment field in the program header contains the default *alignment* value.

CAUTION If a *virtual_address* value is specified, the segment is placed at that virtual address. For the UNIX system kernel this creates a correct result. For files that start via *exec*, this method creates an incorrect *a.out* file because the segments do not have correct offsets relative to their page boundaries.

Mapping directives

A mapping directive tells *ld* how to map input sections to segments. Basically, you name the segment that you are mapping to and indicate what the attributes of a section must be in order to map into the named segment. The set of *section_attribute_values* that a section must have to map into a specific segment is called the entrance criteria for that segment. In order to be placed in a specified segment of the *a.out*, a section must meet the entrance criteria for a segment exactly.

A mapping directive has the following syntax:

```
segment_name : {section_attribute_value}* [ : {file_name}+];
```


For a *segment_name*, you specify any number of *section_attribute_values* in any order, each separated by a space. (At most one section attribute value is allowed for each section attribute.) You can also specify that the section must come from a certain *.o* file(s) via the *file_name* substitutable. The section attributes and their valid values are as follows:

section_name: any valid section name

section_type: \$PROGBITS
 \$SYMTAB
 \$STRTAB
 \$REL
 \$RELA
 \$NOTE
 \$NOBITS

section_flags: ?[!A][!W][!X]

Note the following when entering mapping directives:

- You must choose at most one *section_type* from the *section_types* listed above. The *section_types* listed above are built-in types. For more information on *section_types*, see “Section header” (page 177).
- The *section_flags* values are A for allocatable, W for writable, or X for executable. If an individual flag is preceded by an exclamation mark (!), the linker checks to make sure that the flag is not set. No spaces are allowed between the question mark, exclamation point(s), and the individual flags that make up the *section_flags* value.
- *file_name* may be any valid file name and can be of the form *archive_name(component_name)*, for example:
 /usr/lib/usr/libc.a(printf.o)
 A file name may be of the form **file_name* (see next bullet item). Note that **ld** does not check the syntax of file names.
- If a *file_name* is of the form **file_name*, **ld** simulates a **basename(C)** on the file name from the command line and uses that to match against the mapfile *file_name*. In other words, the *file_name* from the mapfile only needs to match the last part of the file name from the command line. (See “Mapping example” (page 229) for more information.)
- If you use the **-l** option on the **cc** or **ld** command line, and the library after the **-l** option is in the current directory, you must precede the library with **./** (or the entire path name) in the mapfile in order to create a match.

- More than one directive line may appear for a particular output segment, for example, the following set of directives is valid:

```
S1 : $PROGBITS;
S1 : $NOBITS;
```

Entering more than one mapping directive line for a segment is the only way to specify multiple values of a section attribute.

- A section can match more than one entrance criteria. In this case, the first segment encountered in the mapfile with that entrance criteria is used, for example, if a mapfile reads:

```
S1 : $PROGBITS;
S2 : $PROGBITS;
```

the \$PROGBITS sections are mapped to segment S1.

Size-Symbol declarations

Size-symbol declarations let you define a new global-absolute symbol that represents the size, in bytes, of the specified segment. This symbol can be referenced in your object files. A size-symbol declaration has the following syntax:

segment_name @ *symbol_name*

symbol_name can be any valid C identifier, although the `ld` command does not check the syntax of the *symbol_name*.

Mapping example

Example 10-1, “User-defined mapfile” (this page) is an example of a user-defined mapfile. The numbers on the left are included in the example for tutorial purposes. Only the information to the right of the numbers would actually appear in the mapfile.

Example 10-1 User-defined mapfile

```
1      elephant : .bss : peanuts.o *popcorn.o;
2      monkey : $PROGBITS ?AX;
3      monkey : .bss;
4      monkey = LOAD V0x80000000 L0x4000;
5      monkey = LOAD;
6      donkey = ?RWX;
7      donkey : .bss;
8      donkey = ?RX A0x1000;
9      text = ?RWX V0x80008000;
```

Four separate segments are manipulated in this example. The implicitly declared segment `elephant` (line 1) receives all of the `.bss` sections from the files `peanuts.o` and `popcorn.o`. Note that `*popcorn.o` matches any `popcorn.o` file that may have been entered on the `ld` command line; the file need not be in the current directory. On the other hand, if `/var/tmp/peanuts.o` were entered on the `ld` command line, it would not match `peanuts.o` because it is not preceded by a `*`.

The implicitly declared segment `monkey` (line 2) receives all sections that are both `$PROGBITS` and allocatable-executable (`?AX`), as well as all sections (not already in the segment `elephant`) with the name `.bss` (line 3). The `.bss` sections entering the `monkey` segment need not be `$PROGBITS` or allocatable-executable because the `section_type` and `section_flags` values were entered on a separate line from the `section_name` value. (An “and” relationship exists between attributes on the same line as illustrated by `$PROGBITS` “and” `?AX` on line 2. An “or” relationship exists between attributes for the same segment that span more than one line as illustrated by `$PROGBITS` `?AX` on line 2 “or” `.bss` on line 3.)

The `monkey` segment is implicitly declared in line 2 with `segment_type` value `LOAD`, `segment_flags` value `?RWX`, and default `virtual_address`, `physical_address`, `length`, and `alignment` values specified per CPU type. In line 4 the `segment_type` value of `monkey` is set to `LOAD`, `virtual_address` value to `0x80000000` and maximum `length` value to `0x4000`. In line 5 the `segment_type` value of `monkey` is again set to `LOAD` (since the `segment_type` attribute value does not change, no warning is issued).

Line 6 implicitly declares the `donkey` segment. The entrance criteria is designed to route all `.bss` sections to this segment. Actually, no sections fall into this segment because the entrance criteria for `monkey` in line 3 capture all of these sections. In line 8, the `segment_flags` value is set to `?RX` and the `alignment` value is set to `0x1000` (since the `segment_flags` value changes, a warning is issued).

Line 9 changes the `segment_flags` value to `?RWX` and the `virtual_address` value of the text segment to `0x80008000` (since the `segment_flags` value changes, a warning is issued).

The example user-defined mapfile in Example 10-1, “User-defined mapfile” (page 229) is designed to cause warnings for illustration purposes. If you wanted to change the order of the directives to avoid warnings, the example would appear as follows:

Example 10-2 Clean user-defined mapfile

```

1      elephant : .bss : peanuts.o *popcorn.o;
4      monkey = LOAD V0x80000000 L0x4000;
2      monkey : $PROGBITS ?AX;
3      monkey : .bss;
6      donkey = ?RX A0x1000;
5      donkey : .bss;
7      text = V0x80008000;
```

This order eliminates all warnings.

Mapfile option defaults

The **ld** command has three built-in segments (text, data, and note) with default *segment_attribute_values* and corresponding default mapping directives as described under “Segment declarations” (page 225). Even though the **ld** command does not use an actual “mapfile” to store the defaults, the model of a “default mapfile” helps to illustrate what happens when the **ld** command encounters your mapfile.

Example 10-3, “Default mapfile” (page 232) shows how a mapfile would appear for the **ld** command defaults. The **ld** command begins execution behaving as if the mapfile in Example 10-3, “Default mapfile” (page 232) has already been read in. Then **ld** reads your mapfile and either augments or makes changes to the defaults.

CAUTION The **interp** segment, which precedes all others, and the **dynamic** segment, which follows the **data** segment, are not shown in Example 10-3, “Default mapfile” (page 232) and Figure 10-1, “Simple map structure” (page 233) because you cannot manipulate them.

Example 10-3 Default mapfile

```
text = LOAD ?RX;  
text : $PROGBITS ?A!W;  
  
data = LOAD ?RWX;  
data : $PROGBITS ?AW;  
data : $NOBITS ?AW;  
  
note = NOTE;  
note : $NOTE;
```

As each segment declaration in your mapfile is read in, it is compared to the existing list of segment declarations as follows:

1. If the segment does not already exist in the mapfile, but another with the same *segment-type* value exists, the segment is added before all of the existing segments of the same *segment_type*.
2. If none of the segments in the existing mapfile has the same *segment_type* value as the segment just read in, then the segment is added by *segment_type* value to maintain the following order:
 1. INTERP
 2. LOAD
 3. DYNAMIC
 4. NOTE
3. If the segment is of *segment_type* LOAD and you have defined a *virtual_address* value for this LOADable segment, the segment is placed before any LOADable segments without a defined *virtual_address* value or with a higher *virtual_address* value, but after any segments with a *virtual_address* value that is lower.

As each mapping directive in your mapfile is read in, the directive is added after any other mapping directives that you already specified for the same segment but before the default mapping directives for that segment.

Internal map structure

One of the most important data structures in the ELF-based **ld** is the map structure. A default map structure, corresponding to the model default mapfile mentioned above, is used by **ld** when the command is executed. Then, if the mapfile option is used, **ld** parses the mapfile to augment and/or override certain values in the default map structure.

A typical (although somewhat simplified) map structure is illustrated in Figure 10-1, "Simple map structure" (this page). The "Entrance Criteria" boxes correspond to the information in the default mapping directives and the "Segment Attribute Descriptors" boxes correspond to the information in the default segment declarations. The "Output Section Descriptors" boxes give the detailed attributes of the sections that fall under each segment. The sections themselves are in circles.

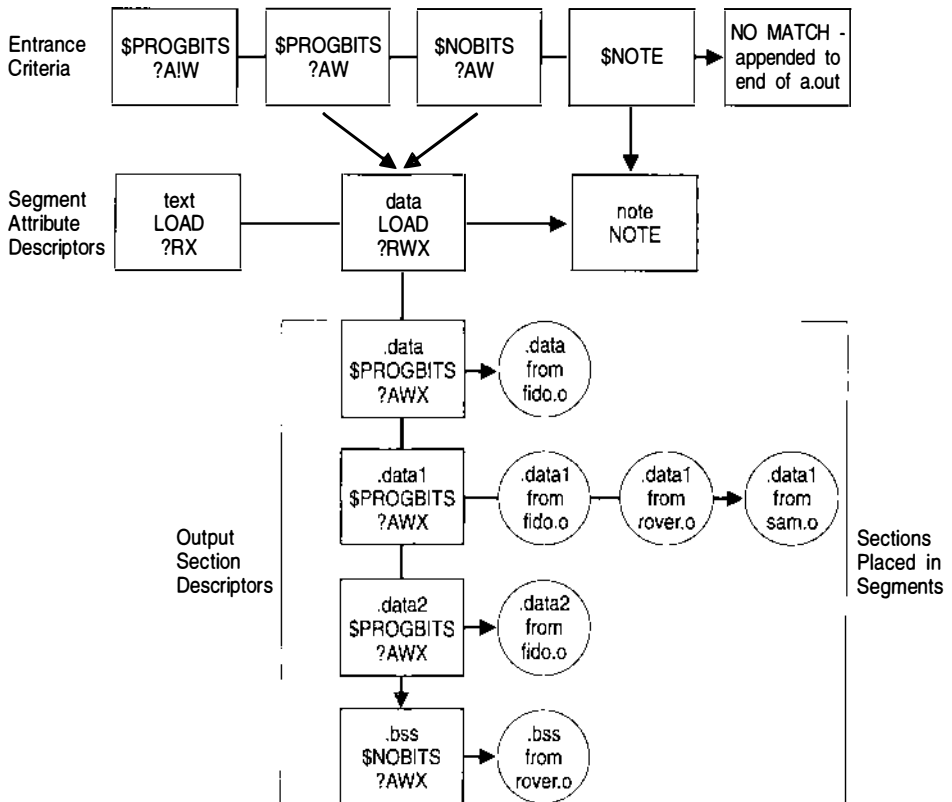


Figure 10-1 Simple map structure

ld performs the following steps when mapping sections to segments:

1. When a section is read in, **ld** checks the list of Entrance Criteria looking for a match. (All specified criteria must match):
 - In Figure 10-1, “Simple map structure” (page 233), for a section to fall into the text segment it must have a *section_type* value of \$PROGBITS and have a *section_flags* value of ?A!W. It need not have the name `.text` since no name is specified in the Entrance Criteria. The section may be either X or !X (in the *section_flags* value) since nothing was specified for the execute bit in the Entrance Criteria.
 - If no Entrance Criteria match is found, the section is placed at the end of the *a.out* file after all other segments. No program header entry is created for this information. See Chapter 8, “ELF object files” (page 169) for information on program headers.
2. When the section falls into a segment, **ld** checks the list of existing Output Section Descriptors in that segment as follows:
 - If the section attribute values match those of an existing Output Section Descriptor exactly, the section is placed at the end of the list of sections associated with that Output Section Descriptor.
 For instance, a section with a *section_name* value of `.data1`, a *section_type* value of \$PROGBITS, and a *section_flags* value of ?AWX falls into the second Entrance Criteria box in Figure 10-1, “Simple map structure” (page 233), placing it in the data segment. The section matches the second Output Section Descriptor box exactly (`.data1`, \$PROGBITS, ?AWX) and is added to the end of the list associated with that box. The `.data1` sections from *fido.o*, *rover.o*, and *sam.o* illustrate this point.
 - If no matching Output Section Descriptor is found, but other Output Section Descriptors of the same *section_type* exist, a new Output Section Descriptor is created with the same attribute values as the section and that section is associated with the new Output Section Descriptor. The Output Section Descriptor (and the section) are placed after the last Output Section Descriptor of the same *section_type*. The `.data2` section in Figure 10-1, “Simple map structure” (page 233) was placed in this manner.

- If no other Output Section Descriptors of the indicated *section_type* exist, a new Output Section Descriptor is created and the section is placed so as to maintain the following *section_type* order:

```
$DYNAMIC
$PROGBITS
$SYMTAB
$STRTAB
$RELA
$REL
$HASH
$NOTE
$NOBITS
```

The `.bss` section in Figure 10-1, “Simple map structure” (page 233) illustrates this point.

NOTE If the input section has a user-defined *section_type* value (that is, between `SHT_LOUSER` and `SHT_HIUSER`) it is treated as a `$PROGBITS` section. No method exists for naming this *section_type* value in the mapfile, but these sections can be redirected using the other attribute value specifications (*section_flags*, *section_name*) in the entrance criteria. For more information, see “Section header” (page 177).

3. If a segment contains no sections after all of the command line object files and libraries have been read in, no program header entry is produced for that segment.

NOTE Note that input sections of type `$SYMTAB`, `$STRTAB`, `$REL`, and `$RELA` are used internally by `ld`. Directives that refer to these *section_types* can only map output sections produced by `ld` to segments.

Error messages

When using the mapfile option, `ld` can return the following types of error messages:

Warnings	Do not stop execution of the linker nor prevent the linker from producing a viable <i>a.out</i> .
Fatal Errors	Stop execution of the linker at the point at which the fatal error occurred.

Either `warning:` or `fatal:` appears at the beginning of each error message. Error messages are not numbered.

Warnings

The following conditions produce warnings:

- A *physical_address* or a *virtual_address* value or a *length* value appears for any segment other than a `LOAD` segment (the directive is ignored).
- A second declaration line exists for the same segment that changes an attribute value(s) (the second declaration overrides the original).
- An attribute value(s) (*segment_type* and/or *segment_flags* for text and data; *segment_type* for note) was changed for one of the built-in segments.
- An attribute value(s) (*segment_type*, and/or *segment_flags*) was changed for a segment created by an implicit declaration.

Fatal errors

The following conditions produce fatal errors:

- Specifying more than one **-M** option on the command line.
- Specifying both the **-r** and the **-M** option on the same command line.
- Specifying the **-M** option without the **-dn** option on the command line (**-dy** is the default; you must specify **-dn** with the **-M** option).
- A mapfile cannot be opened or read.
- A syntax error is found in the mapfile.

NOTE `ld` does not return an error if a *file_name*, *section_name*, *segment_name* or *symbol_name* does not conform to the rules under “Mapfile structure and syntax” (page 224) unless this condition produces a syntax error. For instance, if a name begins with a special character and this name is at the beginning of a directive line, `ld` returns an error. If the name is a *section_name* (appearing within the directive) `ld` does not return an error.

- More than one *segment_type*, *segment_flags*, *virtual_address*, *physical_address*, *length*, or *alignment* value appears on a single declaration line.
- You attempt to manipulate either the `interp` segment or `dynamic` segment in a mapfile.

CAUTION The `interp` and `dynamic` segments are special built-in segments that you cannot change in any way.

- A segment grows larger than the size specified by a your *length* attribute value.
- A user-defined *virtual_address* value causes a segment to overlap the previous segment.
- More than one *section_name*, *section_type*, or *section_flags* value appears on a single directive line.
- A flag and its complement (for example, A and !A) appear on a single directive line.

Chapter 11

Integrating applications into the Desktop

This paper explains how to use the icon, display resolution, color, and session manager APIs (application programming interfaces) provided with SCO OpenServer.

These APIs help you integrate your application into an SCO system. With them, you can ensure the best look for your application by defining different graphical resources for different display types. Using the SCO APIs will also help make your application's appearance and behavior consistent with that of other software running on an SCO OpenServer system. Users expect and depend on such consistency, because it enables them to learn and use new applications more easily.

You may use any, all, or none of the four APIs described in this paper. We recommend you use all four, because:

- If you use the **icon object API** to put your icon on the Desktop, users can start (launch) your application by double-clicking on your icon. If you do not, users must start your application from the command line (or by finding your executable in its directory window and double-clicking on that file's icon).
- If you use the **display resources API** to adapt to display resolution at run time, the SCO OpenServer system will load those resources that give your application the best appearance on the current display. If you do not, your application may look better on some displays than on others.

- If you use the **color API** to support the Desktop color selector, users can customize your application's color scheme with the SCO graphical **Color** control. If you do not, your application's colors will not change when users globally modify those of the Desktop, Desktop clients, and other applications.
- If you use the **session manager API** to communicate with the session manager, users can terminate a Desktop session while your application is running, then resume the session later with your application restored to the same state it was in at shutdown. If you do not, your application will still be restarted when the session is resumed, but its state at shutdown will not be restored and unsaved data may be lost.

NOTE Of the four listed above, only the icon object API is applicable to character-based applications.

What you should know

This paper assumes that you are familiar with:

- SCO OpenServer
- X Window System Toolkit, Version 11, Release 5
- OSF/Motif® Toolkit, Version 1.2
- *OSF/Motif Style Guide*, Release 1.1
- ICCCM, Version 1.0 (X interclient communications protocols)
- C programming language (ANSI standard)

For more information about these systems and technologies, see the SCO OpenServer Development System documentation, the *Operating System User's Guide* and *Using the Desktop*, and the *Graphical Environment Guide*. The last of these also contains detailed information about rule and icon object script syntax.

NOTE SCO OpenServer Development System includes X Window System® Version 11, Release 5 and the OSF/Motif Toolkit, Version 1.2. Refer to the *Release Notes* for important information about the standards and features in X11R5 and the new Motif Toolkit. Refer to the **Intro(Xm)** manual page for a description of new functionality and enhancements that affect the Motif Toolkit as a whole. See "Backward compatibility" (page 241) for information about running new SCO OpenServer binaries on Open Desktop Release 2.0 and 3.0.

Graphical interface standards and libraries

If your UNIX® application has a graphical interface, it must be able to run in an X Window System environment. The interface should be Motif-compliant, as defined by the *OSF/Motif Style Guide* (Revision 1.1).

It is possible to substantially reduce memory usage by using the X dynamically linked libraries. To take advantage of these performance improvements, use the **-b elf** option when you link your application to the libraries listed in Table 11-1 (this page). Refer to the *Release Notes* for further information about X11R5 and Motif 1.2 performance improvements. Refer to the next section “Backward compatibility” (this page) for information about running new binaries on previous or later versions of SCO systems.

Table 11-1 Libraries to link with

Library	Contains:
Xt	Xt intrinsics
Xm	Motif routines
X11	X Window (Version 11) routines
Xmu	X miscellaneous utilities
malloc	faster version of memory allocation routines
socket	socket interface routines

Refer to Chapter 2, “C compilation system” in the *Programming Tools Guide* for more information about standard COFF libraries, also known as archive libraries, and ELF dynamically linked libraries. For detailed information about shared libraries, see Chapter 14, “Shared libraries” in the *Programming Tools Guide*. Refer to the *Release Notes* for specific information about linking these libraries.

Backward compatibility

Applications that run on Open Desktop Release 1.1, 2.0, and 3.0 will run on the new release of the SCO OpenServer systems; however, the reverse may not be true.

Applications that run on Open Desktop Release 1.1 will run on Open Desktop Release 2.0 and 3.0; however, the reverse may not be true because Release 2.0 and 3.0 add features and technology not available in Release 1.1. Specifically, the APIs discussed in this chapter are new, and so not supported in releases prior to 2.0.

Release 1.1 defined icons through rules. These rules were located with other rules in rule files. When an application was installed, rule files had to be edited so the new rules could be added. The rule files had to be edited again when an application was removed or updated.

Release 2.0 and 3.0 uses a more modular approach: discrete icon objects are installed with a utility, eliminating the need to edit rule files. While icon objects simplify installation of applications onto this and future SCO systems releases, icons defined this way will not display on Release 1.1 Desktops.

For more about installing Release 1.1-compatible applications on a Release 2.0 or 3.0 Desktop, see the *Installation and Upgrade Guide* for those releases. The *Graphical Environment Guide* provides instructions for transferring currently installed and configured applications from Release 1.1 to Release 2.0 or 3.0 Desktops.

Refer to the *Release Notes* for information about Motif 1.2 and Motif 1.1 binary, visual, and behavioral backward compatibility.

Putting your icon on the Desktop

SCO OpenServer system users expect to be able to start your application by double-clicking on its icon. If your application works with text, graphic, or other data files, users also assume that dropping a file icon on your icon will start your application working on that file.

SCO OpenServer system uses an object-oriented approach to defining application icons. A Desktop icon is an object defined by one or more pixmap files (the icon picture), one or more object scripts (that specify what happens when users manipulate the icon), and an object directory (to hold the pixmaps and object scripts). The icon label is taken by default from the name of the object directory. At the beginning of a session, the software looks in designated directories for icon objects, then displays the corresponding icons.

To put your icon on the Desktop:

1. Define the icon graphic (page 243) in four pixmap files: one each for activated and closed, and small versions of the activated and closed icons.
2. Define the actions (page 246) triggered by user manipulation of the icon in object script files.
3. "Install icon files" (page 250) with the **deskconfobj** installation tool.

NOTE The **Object Builder** client allows you to graphically create objects. You can use **Object Builder** to perform step 2 interactively. Using **Object Builder** is generally easier than defining icon triggers manually. Refer to the *Graphical Environment Guide* and the **objbld(XC)** manual page for more information.

You may also:

- Animate icons (page 252).
- Attach pop-up menus to icons (page 248).
- Provide translations of the icon label for different languages (page 253).
- Create special icons for files created by your application (page 255).
- Provide localized versions of pixmaps for different languages.

Create your icon pixmaps

You need to create four icon pixmaps for each client. These are:

Filename	Purpose
picture.px	closed/standard icon
open.px	open/activated icon
s_picture.px	small closed/standard icon
s_open.px	small open/activated icon.

The easiest way to prepare an icon pixmap is to start with an existing icon:

1. Select a Desktop icon to use as a base template.

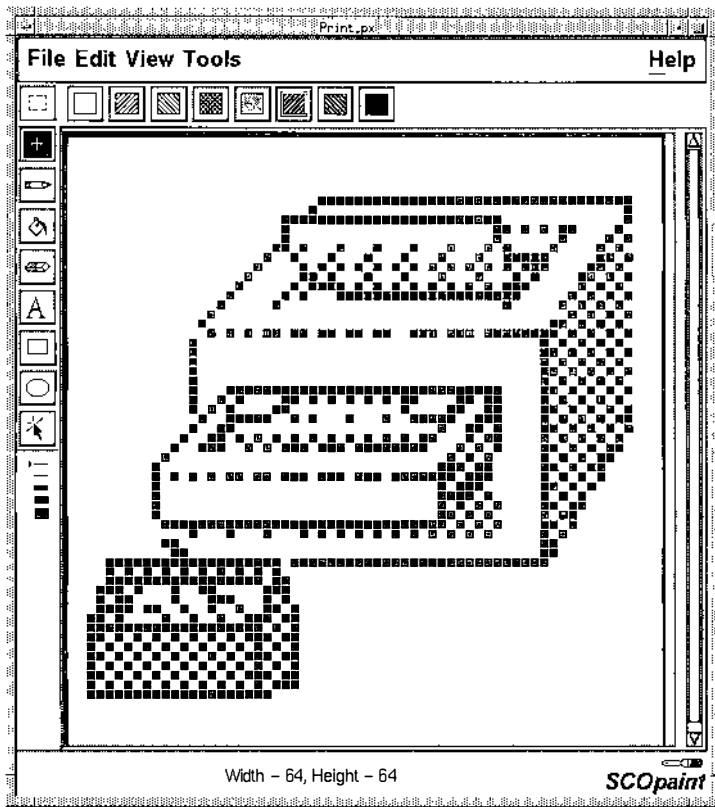
Choose any existing Desktop icon from one of the toolsheds, or by opening the *bitmaps* directory in the Desktop tree.

2. Drop that icon on the Paint icon.

This opens the **Paint** accessory with the icon you selected displayed, as shown below. If you see the following message, click on **Yes**:

```
... does not seem to be a pixmap file.
Do you want to edit the icon picture?
```

NOTE To edit color icons with the **Paint** accessory, you must work on a color monitor with a server that supports up to 256 colors.



We recommend you use the **Paint** accessory (located in the Desktop Accessories window) to create your icon. Although you may use any icon editor or graphics program that can save to a file in either XPM2 (Version 2.8) or XPM3 (Version 4.3) pixmap format, only pixmaps created with the **Paint** accessory can use the transparent “color” needed to create borderless icons. Instructions for using the **Paint** accessory are available online through **Help**.

3. Edit the image into the picture you want.

This technique ensures that your pixmap will be based on a large pixel grid, and that it uses appropriate colors.

The Desktop can be run on servers that support as few as 16 colors. Colors are identified by their hexadecimal values at the beginning of a pixmap file (and can be viewed with a text editor). To ensure that your icon will look good even when displayed by 16-color servers, use the same colors as other Desktop icons do:

	R	G	B	
icon color 1:	# FFFF	FFFF	0000	(Yellow)
icon color 2:	# FFFF	0000	0000	(Red)
icon color 3:	# 0000	0000	FFFF	(Blue)
icon color 4:	# 0000	AAAA	0000	(Green)
icon color 5:	# 8207	8207	8207	(Dark Grey)
icon color 6:	# C30B	C30B	C30B	(Light Grey)
black				
white				
transparent				

When you use one of the Desktop icons as a starting point, these colors are already selected for you.

Blend additional colors by stippling these together in various combinations. If you want to adapt an icon you have created elsewhere, use the **Paint** accessory's **Grab Color** feature to capture colors from a Desktop icon and transfer them to your icon.

The transparent "color" lets the background color of the Desktop (or directory window) show through, so you can create borderless icons. Generally, an icon can be more readily picked out of a group of distinctly shaped icons than it can from a group of icons with identical rectangular borders.

Do not include the icon label in the pixmap. The label is defined by the name of the icon object directory.

4. Save the edited image.

Save the edited image in a pixmap file named *picture.px*.

Note that if you have an existing icon that you had previously created in black-and-white bitmap format, you may use it, as long as you rename the bitmap file *picture.px*.

5. Test your icon's appearance on various servers and displays.

Your icon will be seen on a variety of displays. View your icon on different displays to verify that it is legible in various resolutions and on grayscale displays. To get the best look on monochrome displays, use the **Paint** accessory to explicitly map each color you use to black or white. Refer to the **scopaint(XC)** manual page for more information.

Define icon triggers

The action of pressing a mouse button while pointing to an icon is called an icon trigger, because it triggers an action associated with the icon. Table 11-2 shows the default icon triggers.

Table 11-2 Icon triggers

User action	Mouse button	Trigger type	Object script file name	Used for
double-click	1	static	activate	start
double-click	2	static	alt_activate	alternate start
hold down	3	hold	menu	pop-up menu
drag and drop	1	drop	drop	start on file
drag and drop	2	drop	drop	start on file
drag and drop	<Ctrl> 1	drop	alt_drop	alternate start on file
drag and drop	<Ctrl> 2	drop	drop	alt_drop

On a three-button mouse, the buttons are referred to as mouse buttons 1, 2 and 3. Mouse buttons 1 (the left button by default) and 3 (the right button by default) can be swapped to accommodate left-handed users. On a two-button mouse, users can simulate button 2 by pressing both buttons simultaneously. For more about how a mouse is used, see *Using the Desktop*.

To define an icon trigger:

1. Write a Deskshell object script for each trigger you want to define.

You should only define actions for the triggers listed in Table 11-2 (this page). Other user manipulations of your icon (single-clicking on it or dragging it to a new location, for example) are handled automatically. Sample object scripts for activate, drop, and menu triggers are presented on the following pages.

The activate trigger must be used to start your application. By default, double-clicking with mouse button 2 (the alt_activate trigger) brings up a a dialog box that prompts the user for command line options to the startup command in your *activate* script. Similarly, a file icon dragged with mouse button 3 and dropped on your icon (the d3 trigger) brings up

a prompt for startup options, then executes your script for the drop trigger.

Use the Desksell command language to write object scripts. It provides the same basic functions as the Bourne shell, along with additional commands for defining special Desktop actions. Instructions for using Desksell are in the chapter “Writing scripts in Desksell” in the *Graphical Environment Guide*.

Only write object scripts for those triggers that are relevant to your application. For example, an application that simply displays a clock does not need to define what happens when a text file is dropped on it. If a trigger is not defined with an object script, a message will notify the user that the action (double-click, hold, or drop) is not defined for that button.

You can use a similar application’s trigger files as templates for yours, by copying them to the desired filenames. In doing so, you will include default rules for 2-state animation.

2. Name each object script file as specified in Table 2.

When assigning functions to mouse buttons, follow the guidelines defined in the *OSF/Motif Style Guide* regarding which button does what. Your application will be easier to learn if it uses the mouse buttons in the same ways other applications do.

Starting your application with the activate trigger

Your *activate* object script should start your application when a user double-clicks on your icon with mouse button 1. This script must be named *activate*.

For example, an *activate* script to start a graphical clock could be as simple as:

```
1      begin
2      /usr/bin/X11/xclock
3      finish
```

If you use a relative pathname in your object scripts, you must be certain that users have the correct path set in their environment. A safer approach is to use an absolute path as shown in the previous example.

To start a character-based application with its icon, the *activate* script must open a DOS or UNIX (**scoterm**) window in which to run the application. The following Desksell object script starts a UNIX editor, **vi**, in a **scoterm** window.

```
1      begin
2      shell -t 'Editor' vi
3      finish
```

The **begin** and **finish** statements cause the displayed icon to change from closed to activated, if you created the appropriate pixmap files (page 243).

The **shell** statement is a Desktop resource that defines the terminal emulation; it is set to **scoterm** by default. The **-t** option passes the title to the **scoterm** window. You can also localize (page 253) this title and other messages based on the **\$LANG** environment variable.

For more about the UNIX window, see **scoterm(XC)**. For more about the DOS window, see *SCO Merge User's Guide*.

Starting your application with the drop trigger

If your application uses text, graphic, or other data files, users expect that dropping a file icon on your icon will start your application working on that file. For example, dropping a text file on an editor's icon should start the editor with the text file open and ready to edit.

The following *drop* object script starts a text editor and passes to the editor the name of the file dropped on it. The system sets the variable **\$dynamic_args** to the filename(s) represented by the dropped icon(s). (For more about Desktop variables, see the *Graphical Environment Guide*.)

```
begin
shell -t 'Editor' vi $dynamic_args
finish
```

The **begin** and **finish** statements cause the displayed icon to change from closed to activated, if you created the appropriate pixmap files (page 243).

Users can drop a group of icons on your application's icon. Consider how your application will handle this. For example, should the text editor open an editor window for each icon dropped? Or should it open one file and place the others in a queue? If more than one approach could be useful, let users choose the default behavior.

Popping up a menu with the menu trigger

Only button 3 can be used as a popup menu trigger. If defined, the menu trigger must display a pop-up menu.

For example, this *menu* script pops up the menu **PopupClockMenu** (which is used to start **xclock** with different startup options):

```

1      rule='
2      menu: PopupClockMenu
3      {
4          menu_item
5          {
6              title='digital'
7              action
8              {
9                  begin
10                 xclock -digital
11                 finish
12             }
13         }
14         menu_item
15         {
16             title='analog'
17             action
18             {
19                 begin
20                 xclock
21                 finish
22             }
23         }
24         dividing_line;
25         menu_item
26         {
27             title='help'
28             action
29             {
30                 help 'static_arg'
31             }
32         }
33         remove_dynamic_rule PopupClockMenu
34         dynamic_rule PopupClockMenu $rule
35         popup PopupClockMenu $static_arg

```

lines 4-23 Define the **digital** and **analog** menu items. Each item causes the **xclock** executable to be run with the appropriate option.

line 24 Creates a horizontal line between the first two menu items and the last.

lines 25-32 Defines the **help** menu item.

lines 33-35 Removes any old **PopupClockMenu**, creates a new dynamic **PopupClockMenu**, and creates the popup menu.

Install icon files

The approach to icon definition depends on correct installation of your pixmap, object scripts, and rule files. Use the **deskconfobj** object installation tool in your installation script to ensure that your icon files are installed properly. Your installation script tells **deskconfobj** what to install, and **deskconfobj** determines where.

In this release, pixmaps and object scripts are copied into a directory named *label.obj*, where *label* is the label that appears beneath your icon. This directory is created in the */usr/lib/X11/XDesktop3/applications* directory. Application rules are currently installed in the */usr/lib/X11/XDesktop3/C.xdt/apprules* directory or in the */usr/lib/X11/XDesktop3/lang.xdt/apprules* directory, depending on whether you install localized versions of your rules. These directories may change in future releases, but if you use **deskconfobj** to install your icon object, your installation script will still work.

Because ordinary users usually do not have permission to copy files into the required directories, **deskconfobj** can only be used if the installing administrator is logged in as *root*. If you want your application to be installable by users in their directories, see “Making applications user-installable” (page 251).

NOTE If you will be installing your application with the **custom(ADM)** installation utility, put the following commands in the **POST_CONFIGURE** section of the application’s **ccs** script.

In your installation script:

1. **Reset the file-creation mode mask with:**

```
umask 002
```

2. **Temporarily add */usr/lib/X11/IXI/XDesktop/bin/i3sc0322* to the search path.** Alternatively, you can prepend this path to each **deskconfobj** command.
3. **Install icons and object scripts with this syntax:**

```
deskconfobj -aobj -name label -src source_directory
```

Use the **-aobj** (add object) option to install a new object on the Desktop. For *label*, substitute the text that will be displayed beneath your icon. You may use the name of your application for the label, or any other string that is allowed in a UNIX directory name. For *source_directory*, substitute the path to the directory from which icon pixmaps and object scripts are to be copied. All files in the source directory will be copied into the */usr/lib/X11/XDesktop3/applications/label.obj* directory.

4. Install application rules (if any) with this syntax:

```
deskconfobj -arule -name rule_name -src file
```

Use the **-arule** (add rule) option to add a rules file associated with your object. For *rule_name*, substitute a name that is unlikely to be duplicated by other programs (typically your application's class name). For *file*, substitute the pathname of the file that contains the rules to be copied. Most applications should put all of their application rules in a single source file (or in a series of localized source files with the same name).

NOTE Newly installed rules will not be recognized until the system is restarted.

5. Remove `/usr/lib/X11/IXI/XDesktop/bin/i3sc0322` from the search path.

Skip this step if you prepended this path to the commands instead of adding it to the search path in step 1.

Removing pixmaps, object files, and application rules

You may also use **deskconfobj** in your “de-installation” script to cleanly remove files it has installed and icon object directories it has created.

To remove an icon object, use this syntax:

```
deskconfobj -dobj -name label
```

This removes the entire icon object, including the *label.obj* directory and all the pixmaps and object scripts in it.

To remove an application rule file, use this syntax:

```
deskconfobj -drule -name rule_name
```

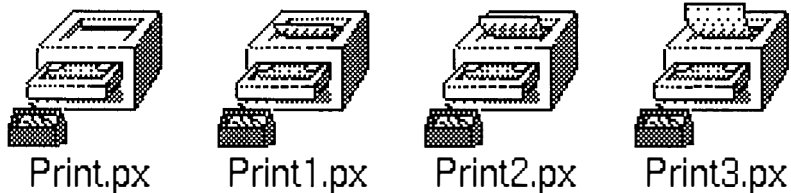
Making applications user-installable

The **deskconfobj** tool cannot be used to install an application into a user's directory. You can install an icon object in a user's directory tree by creating an *label.obj* directory and copying your pixmap(s) and object script(s) into it. Your icon will appear in the *label.obj* directory's parent directory's window.

On the SCO OpenServer system, all user customization is stored in a special subdirectory `$HOME/.xdt_dir`. Your application's rules should be placed in `$HOME/.xdt_dir/modules/appname.auto`. The *.auto* suffix causes this rule to be read each time the Desktop is started; *appname* is the name of your application, and can be any legal filename.

Animating icons

To animate your icon, present a series of icon pictures in sequence. The **Print** icon on the Desktop is an animated icon; when a file icon is dropped on it, paper slides out of the printer. The **Print** icon is animated by stepping through this series of pictures:



You can define an animated icon just like any other icon, except that you must supply more than one pixmap, along with directions for when to display each pixmap.

To create an animated icon:

1. Create the pixmaps (page 243).

The example used to illustrate these instructions defines an appointment book icon that opens when a **calendar** program is started. This icon picture is made of four pixmaps: closed book (*picture.px*), book one-third open (*Calendar1.px*), book two-thirds open (*Calendar2.px*), and open book (*Calendar3.px*).

2. Add animation to the *activate* script.

For example:

```
4      animate $static_arg true $static_arg/Calendar1.px \
5      $static_arg/Calendar2.px $static_arg/Calendar3.px

6      /usr/bin/X11/calendar -monthly $*
```

line 4 Displays the pixmaps, (files ending in *.px*), in the sequence indicated. The *animate* sequence should be a single line. The **true** option resets the icon to its original state after the animated sequence.

line 5 Starts the application. *\$** includes any options the user enters in the dialog box displayed by the *alt_activate* trigger (page 246).

For more about object scripts, see “Define icon triggers” (page 246).

Animation can also be used to convey information about the state of your application. For example, an icon for a mail program could change its appearance to indicate that new mail has arrived. If you want to indicate the state change when an icon trigger is activated, test for the state and

declare the variation class in the appropriate object script. To indicate the state change without waiting for a trigger, your application must test for the state change and send instructions directly to the Desktop (see the `xdt3(XC)` manual page and the *Graphical Environment Guide*).

3. **Install the pixmaps, rules, and object scripts** (page 250) **with the `deskconfobj` tool.**

The pixmaps and scripts will be installed in the icon object directory. By default, your rule file will be copied into the `/usr/lib/X11/IXI/XDesktop/rules/modules` directory.

Localizing icon labels and messages

The environment variable `$LANG` sets a user's language preference. The presence and value of this variable determine how icon labels and messages are displayed during a user's session.

By default, your icon's label is taken from the name of your icon's object directory. If the directory is named *Calendar.obj*, by the icon will be labeled "Calendar". If the file *title* exists in the object directory, the contents of that file are used as the title.

Messages, such as menu titles and error messages, are often explicitly defined in the trigger files as absolute values, such as:

```
title='analog clock'
```

In many cases, this default behavior is undesirable, such as when you want to be able to easily change messages and labels to another language. To do so, you must:

1. Modify the rules found in the *activate*, *drop*, and *menu* trigger files to include a label and message variables rather than absolute values.
2. Create or modify the *text* file associated with the specified language to define label and message variables.

To modify the trigger file:

1. **Change the absolute label name** to a variable called `$t1`, and absolute message names to variables called `$m1`, `$m2`, and so on.

For example, the *menu* example (page 248) could be modified to localize messages:

```

1      rule='
2      menu: PopupClockMenu
3      {
4          menu_item
5          {
6              title='$m1'
7              action
8              {
9                  begin
10                 xclock -digital
11                 finish
12             }
13         }
14         menu_item
15         {
16             title='$m2'
17             action
18             {
19                 begin
20                 xclock
21                 finish
22             }
23         }
24         dividing_line;
25         menu_item
26         {
27             title='$m3'
28             action
29             {
30                 help 'static_arg'
31             }
32         }
33     remove_dynamic_rule PopupClockMenu
34     dynamic_rule PopupClockMenu $rule
35     popup PopupClockMenu $static_arg

```

lines 6, 16, 27

These lines contain the new message variables to be localized in the *object.obj/\$LANG/text* file.

To modify the text file:

1. **Ensure that the appropriate language directory exists in the object directory.**

For example, if you are localizing a Spanish language application for use in Spain, ensure that the *spanish_SPAIN* directory exists in the object directory.

2. Create a file called *text* to include the messages and labels.

Or, if the file already exists, edit it. Here is a sample *text* file:

```
t1='Calendario'
m1='Enero'
m2='Febrero'
.
.
.
m12='Diciembre'
```

In this sample file, the title (t1) is set to the Spanish translation of Calendar, and each message (m1-m12) corresponds to a month of the year.

Creating icons for user data files

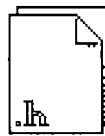
You can use application rules to define special icons for certain types of files, and to start your application when a user double-click on one of those icons. For example, specially formatted calendar files created and displayed by a **calendar** program could be represented by a calendar file icons.

To define special icons for data files:

1. Create a pixmap for the file icon (page 243).

The name of the pixmap file must end in the *.px* extension.

To help users identify it as a *file* icon, you should base the design of your data file icon on the standard SCO OpenServer system file icon. For example, a C header file is represented by a file icon labeled with an ".h":



hfile.px

2. Write a rule that assigns the icon to a type of file.

For example, the following rule specifies that files with the *.cal* suffix will be represented by the icon pixmap file *Cal_file.px*. The rule also specifies that double-clicking on one of these file icons will have the same effect as dropping the file icon on the application's icon: it will start the **calendar** program with the calendar file displayed. For more about Desktop rules, see the *Graphical Environment Guide*.

```

1 icon_rules
2 {
3     *.cal /F
4     {
5         picture=Cal_file.px;

6         trigger_action: activate
7         {
8             filetype /usr/bin/calendar
9             if [ == $_type(1) F ] && [ == $_type(2) X ]
10            then
11                /usr/bin/calendar $* $static_arg
12            else
13                fyi -t 'Calendar' 'Calendar is not installed on your system'
14            fi
15        }
16    }
17 }
```

- line 1 Identifies the start of the **icon_rule** section of your rule file.
- lines 3-5 Specify that files with names ending in *.cal* will display the *Cal_file.px* pixmap. The system searches certain directories below */usr/lib/X11/IXI/XDesktop/bitmaps* for the file.
- lines 6-15 Define what happens when the user double-clicks (the activate or drop trigger) on the file icon.
- lines 8-9 Check the file type of */usr/bin/calendar*. If it is an executable, do line 11. If not, do line 13.
- line 11 Executes the **calendar** program. *\$** includes any arguments the user enters in the drop trigger dialog box. The filename(s) represented by the dropped icon(s) are substituted for *\$static_arg*.
- line 13 Prints an error message if */usr/bin/calendar* is not executable. The message assumes that not having execute permission for the file is, for the Desktop user, functionally equivalent to its not being installed.

3. Use the `deskconfobj` tool to install your pixmap and rule.

Use the following syntax to install your pixmap:

```
deskconfobj -alrg -name pixmap_name -src file
```

Use the **-alrg** (add large pixmap) option to add a pixmap. For *pixmap_name*, substitute the name of the pixmap file (*Cal_file.px* for the above example). For *file*, substitute the pathname of the pixmap file to be copied.

Use the following syntax to install your rule:

```
deskconfobj -arule -name rule_name -src file
```

For *rule_name*, substitute a name that is unlikely to be duplicated by other programs (typically your application's class name). For *file*, substitute the pathname of the file that contains the rules to be copied.

NOTE Newly installed rules will not be recognized until the system is restarted.

The data file's name will be used as the icon label. For more about using `deskconfobj`, see "Install icon files" (page 250).

Adapting to display resolution at run time

The SCO OpenServer system is designed to run on a wide variety of hardware platforms, with resolutions ranging from 640-by-480 pixels to beyond 1280-by-1024 pixels. Visual resources defined for a specific display resolution can produce inappropriately sized, unpleasant, or even illegible results when your application is displayed at a different resolution.

You may specify a different set of visual resources for each display resolution or type with an ODR (display resources) file. In an ODR file, standard **#if**, **#elif**, and **#else** preprocessor directives are used to test the values of certain display attributes, such as **WIDTH**. For example, the following tests the **WIDTH** of the display (in pixels) and specifies font sizes accordingly.

```

    #if WIDTH < 800
        TypicalApp*create_button.FontList    -*-helvetica-bold-r*--10-*p-*
        TypicalApp*delete_button.FontList    -*-helvetica-bold-r*--10-*p-*
    #elif WIDTH < 1024
        TypicalApp*create_button.FontList    -*-helvetica-bold-r*--12-*p-*
        TypicalApp*delete_button.FontList    -*-helvetica-bold-r*--12-*p-*
    #elif WIDTH < 1280
        TypicalApp*create_button.FontList    -*-helvetica-bold-r*--14-*p-*
        TypicalApp*delete_button.FontList    -*-helvetica-bold-r*--14-*p-*
    #else
        TypicalApp*create_button.FontList    -*-helvetica-bold-r*--18-*p-*
        TypicalApp*delete_button.FontList    -*-helvetica-bold-r*--18-*p-*
    #endif

```

At the beginning of a session, the session manager uses **xrdb**, the X resource database utility, to check display attributes and resolve the conditional statements in your ODR file. In this way, visual resources appropriate to each display are loaded into the RESOURCE_MANAGER property of the root window.

Although you may define resources for as many display resolutions as you like, we recommend that you at least support the following four resolutions (WIDTH x HEIGHT in pixels):

640 x 480	(VGA)
800 x 600	(SVGA)
1024 x 768	(SVGA)
1280 x 1024	(high-resolution)

Most applications only need to specify resolution-dependent resources in ODR files, but you can test for any display attribute recognized by the **xrdb** utility, shown in Table 11-3 (page 259). If your application does not use the color API (page 261), you may want to define color resources in your ODR file (page 264).

You do not need to define any of the default system color or font resources unless you want to override them. The default font and color resources are defined in `/usr/lib/X11/sco/startup/Fonts` and `/usr/lib/X11/sco/startup/Colors`.

For more about graphical resources, see the *Graphical Environment Guide*. Remember that your resource definitions only affect the content of the window in which your application is running. The window's frame and decorations are controlled by the window manager's resources.

To specify resolution-dependent resources:

1. **Separate resolution-dependent resources from other visual resources.**
Because the ODR files for all installed applications are loaded at the beginning of a session, only define display-dependent resources in ODR files. Define other resources in an *app-defaults* file, as usual (for more about *app-defaults* files, see the *Graphical Environment Guide*).
2. **Define resolution-dependent resources in an ODR file.**
Define all of your display-dependent resources in a single ODR file. Choose a name for your ODR file that is unlikely to be duplicated by other programs (typically your application's class name). Within this file, organize resources according to the type of display with which they are to be used. Use standard **#if**, **#elif**, and **#else** preprocessor directives and **xrdb** display attribute symbols to conditionally specify resources. We use the **WIDTH** (screen width in pixels) symbol to define resolution-dependent resources.

Table 11-3 Display attribute symbols

Symbol	Definition
WIDTH	screen width in pixels
HEIGHT	screen height in pixels
X_RESOLUTION	horizontal screen resolution in pixels/meter
Y_RESOLUTION	vertical screen resolution in pixels/meter
CLASS	GrayScale, PseudoColor, TrueColor, DirectColor, StaticColor, or StaticGray
COLOR	defined only if CLASS = PseudoColor, TrueColor, DirectColor, or StaticColor
PLANES	display depth in bit planes; this value defines the number of available colors
BITS_PER_PLANE	number of significant bits in an RGB color specification; usually unrelated to PLANES

For more about display attributes symbols, see the **xrdb(XC)** manual page.

To avoid conflicts with other applications' ODR resources loaded at the same time, begin the name of each resource in your ODR file with your application's class. For example, an application that displays a calendar might include the following in its ODR file:

```
#if WIDTH < 800
    Calendar.main.width:          300
    Calendar.main.height:        200
#elif WIDTH < 1024
    Calendar.main.width:          400
    Calendar.main.height:        300
#elif WIDTH < 1280
    Calendar.main.width:          500
    Calendar.main.height:        400
#else
    Calendar.main.width:          700
    Calendar.main.height:        500
#endif
```

3. Install your ODR file in the `/usr/lib/X11/sco/startup` directory.

Do this in your installation script.

At the start of a session, the session manager uses the `xrdb` utility to process your ODR file and load the appropriate resources.

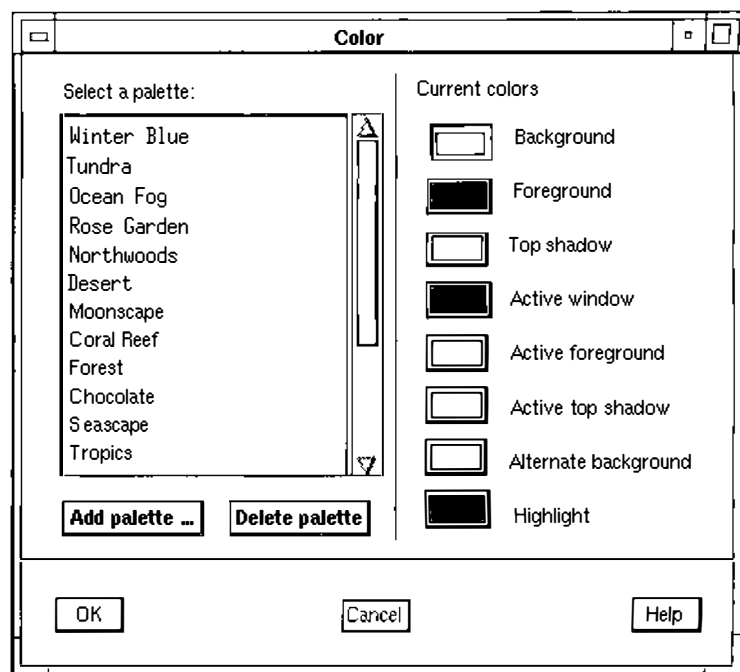
4. Define default display resources in your `app-defaults` file.

To ensure that your application will always display, even if it is run without the session manager (and, therefore, without your ODR file loaded), define default resources in your `app-defaults` file. Include resource definitions for a 1024-by-768-pixel display (the resolution most widely used by the SCO OpenServer system) from your ODR file. Also include the default font resource definitions from `/usr/lib/X11/sco/startup/Fonts` and the default color resource definitions from `/usr/lib/X11/sco/startup/Colors`. Place these default definitions before any overriding definitions of your own. Name your `app-defaults` file with your application's class name, and install it in the `/usr/lib/X11/app-defaults` directory.

When they *are* loaded, your ODR file resource definitions take precedence over the defaults defined in your `app-defaults` file.

Supporting the Desktop color selector

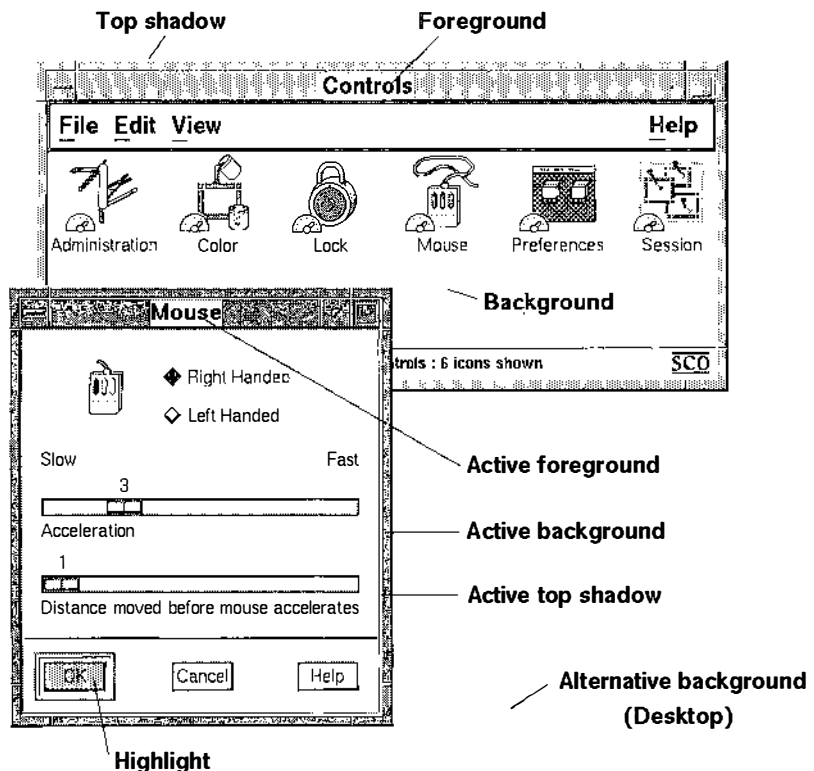
SCO OpenServer system users can use the **Color** utility (located in the Preferences editor) to simultaneously modify the coloring of all Desktop utilities, window frames, menus, and backgrounds. If you specify your color resources through the color API, users will be able to personalize your application's colors along with those of the rest of the Desktop.



The color API is based on palettes. A palette is a coordinated set of eight colors, defined by the current values of the eight color name variables. Each variable defines the color for several graphical objects.

Table 11-4 Color name variables

Variable	Typically used for
scoBackground	window frames, window backgrounds, scrollbar troughs
scoAltBackground	Desktop background, second background color
scoForeground	text
scoTopShadow	top shadows of frames, buttons and other objects
scoActiveBackground	frame of active window
scoActiveForeground	text in frame of active window
scoActiveTopShadow	top shadows in active window
scoHighlight	text or objects that need special emphasis, pressed buttons



The active window is the window with the input focus. The `scoActiveBackground`, `scoActiveForeground`, and `scoActiveTopShadow` variables are used to distinguish the active window from other windows. In the supplied palettes, `scoTopShadow` is lighter than `scoBackground`, and `scoBackground`

contrasts with `scoForeground`. Bottom shadows are always black. When using the color name variables for interface elements other than those listed, consider the relationships of the colors and how that might affect contrast or legibility.

Color palettes are automatically mapped to grayscale displays. Because this might not always yield optimal results, several grayscale palettes are provided. A default monochrome palette mapping is used on monochrome displays.

For more information about the **Color** utility, see the `sccolor(XC)` manual page and the *Graphical Environment Guide*.

To support the Desktop color selector:

1. Use color name variables to define color resources.

In your *app-defaults* file, define color resources with the color name variables listed in Table 11-4 (page 262). The system will set the variables to the corresponding color definitions from the currently selected palette.

For example, an application that displays a calendar might include the following resource definitions in its *app-defaults* file:

```
*datebox.back.background:      scoAltBackground
*current.week.background:      scoActiveForeground
*meeting.alarm.background:     scoHighlight
```

Name your *app-defaults* file with your application's class name, and install it in the `/usr/lib/X11/app-defaults` directory. For more about *app-defaults* files, see the *Graphical Environment Guide*.

2. Link your application with the new release of the SCO OpenServer Development System.

Color name variables used by applications that have *not* been linked with this release of the SCO OpenServer system will be mapped to the default palette, Winter Blue. When a user changes Desktop colors by selecting a new palette with the **Color** utility, your application's colors will continue to be defined by the Winter Blue palette.

3. Include the default color resources in your *app-defaults* file.

To ensure that your application will always display, even if it is run without the session manager, copy into your *app-defaults* file the default color resource definitions from `/usr/lib/X11/sco/startup/Colors` (listed below). Place these definitions before any overriding definitions of your own.

*Background:	scoBackground
*Foreground:	scoForeground
*topShadowColor:	scoTopShadow
*bottomShadowColor:	Black
*activeBackground:	scoActiveBackground
*activeForeground:	scoActiveForeground
*activeTopShadowColor:	scoActiveTopShadow
*activeBottomShadowColor:	Black
*troughColor:	scoBackground
*armColor:	scoHighlight
*highlightColor:	scoForeground
*selectColor:	scoForeground
*borderColor:	Black

Defining display-dependent colors

If your application makes extensive or special use of color, you may want to define your color resources directly rather than through the color API's color name variables.

NOTE The following applies only to those color resources that are not defined with the color API color name variables (page 262).

When defining color resources directly, consider the wide variety of displays on which the Desktop is viewed. Resources defined for a color display may not work for a monochrome or grayscale display, and some servers support more colors or shades of gray than do others.

If your colors work well on all supported displays (color, monochrome, grayscale) and servers (16-color and above), define your color resources in your *app-defaults* file. If you want to use different colors for different displays or servers, conditionally define your color resources in your ODR file. For more about preparing an ODR file, see "Adapting to display resolution at run time" (page 257).

Define color resources in your ODR file *only* if your application makes special use of color that cannot be achieved with the color name variables. You may explicitly define colors for certain objects and use the color API for the rest. This approach is useful when you want to prevent users from changing the color of an object (the red, green, and blue slider bars of an RGB color mixer, for example).

Define display-dependent resources as you would resolution-dependent ones: use standard **#if**, **#elif**, and **#else** preprocessor directives and **xrdb** display attribute symbols (page 259) to conditionally specify resources.

Use the **xrdb** COLOR symbol to differentiate color and monochrome displays.

```
#if COLOR
    . . .
    [definitions for color displays]
    . . .
#else
    . . .
    [definitions for monochrome displays]
    . . .
#endif
```

For most applications, defining separate resources for color and monochrome displays should be adequate. Check how your color resources look on a grayscale display. If the result is unacceptable, use the CLASS symbol to define grayscale resources separately.

```
#if CLASS = GrayScale
    . . .
    [definitions for grayscale displays]
    . . .
#elif COLOR
    . . .
    [definitions for color displays]
    . . .
#else
    . . .
    [definitions for monochrome displays]
    . . .
#endif
```

NOTE Your SCO OpenServer system or X terminal may be running on a server that only supports 16 colors (or shades of gray). The 16 color cells are taken by the palette manager (eight color cells), the server (two color cells, for black and white), and the Desktop icons (six color cells). If you need to define more colors, you must install your own colormap using the facilities provided by the Xlib library.

Use the PLANES symbol to specify resources for 16-color (or grayscale) servers.

```
#if CLASS = GrayScale
    #if PLANES < 8
        . . .
        [definitions for 16-grayscale displays]
        . . .
    #else
        . . .
        [definitions for 256(or more)-grayscale displays]
        . . .
    #endif
#elif COLOR
    #if PLANES < 8
        . . .
        [definitions for 16-color displays]
        . . .
    #else
        . . .
        [definitions for 256(or more)-color displays]
        . . .
    #endif
#else
    . . .
    [definitions for monochrome displays]
    . . .
#endif
```

In your *app-defaults* file, define default values for resources defined in your ODR file. These default definitions will ensure that your application will always display, even if it is run without the session manager (and, therefore, without your ODR file loaded).

Communicating with the session manager

SCO OpenServer system users expect to be able to end a session without first terminating applications. When the session is resumed, users expect to see applications exactly where they were at shutdown. The session manager is responsible for gracefully shutting down all applications at the end of a session, then restarting them when a new session begins. Your application should supply the session manager with the information needed to restart it in the state it was in when the user logged out.

Restart information is conveyed to the session manager through the **WM_COMMAND** property placed on an application's top-level window. Before ending a session, the session manager will send a **WM_SAVE_YOURSELF** message to applications. When your application receives this message, it must immediately prepare itself for shutdown by placing itself in an appropriate state, then setting the **WM_COMMAND** property to a command that will restart the application and restore it to as similar

a state as possible. No interaction with the user is permitted during this process. When the next session begins, the session manager will restart your application with the specified command. It will also restore the application's geometry and window state (iconified or maximized) if the application does not.

For example, after receiving a **WM_SAVE_YOURSELF** message, a text editor might save edits made to the file *letter* since the last user-initiated save in a temporary file named *letter.tmp*. It could then use **XSetCommand(XS)** to set **WM_COMMAND** to something like **editor -rt letter.tmp**. When the session is resumed, the session manager will restart the editor with this command. The editor has defined the **-rt** option to read in the indicated temporary file, and perhaps display a message that there are unsaved edits left over from the previous session. Note that the editor could not ask the user whether it should save the unsaved edits, because no interaction with the user is permitted between receiving the **WM_SAVE_YOURSELF** message and shutdown.

If your application does not communicate with the session manager, the session manager will simply disconnect your application from the server at shutdown. Unsaved data may be lost. When the session is resumed, your application will be restarted with the command with which it was last started, and with the geometry and window state (iconized or maximized) it had at shutdown. Its state at shutdown will not be restored.

The session manager will restore a DOS or UNIX terminal window with the command that opened it. If a character-based application was started inside one of these windows after the window was opened, the application will not be restored. If the application was started as an argument to the **scoterm** command, it will be restarted with that same argument. The application's state at shutdown will not be restored.

To communicate with the session manager:

1. Use standard Xt intrinsics calls to start your application.

When you use standard Xt intrinsics calls to start your application, the properties **WM_COMMAND** and **WM_CLIENT_MACHINE** are placed on your top-level window, **WM_COMMAND** is set to your startup command, and **WM_CLIENT_MACHINE** is set to the name of the machine on which your application is running.

2. Place the atom WM_SAVE_YOURSELF in the WM_PROTOCOLS property on your application's top-level window.

To do this, include the following in your application's code.

```
1 #include <X11/Intrinsic.h>
2 #include <X11/Protocols.h>
3 #include <X11/AtomMgr.h>
4 #include <X11/Xatom.h>

5 void
6 SetupSMProtocols(toplevel)
7 Widget toplevel;
8 {
9     extern void    SaveYourselfCB();

10     Atom    protocols[1];
11     Atom    xa_WM_SAVE_YOURSELF;

12     xa_WM_SAVE_YOURSELF = XInternAtom(XtDisplay(toplevel),
13                                     "WM_SAVE_YOURSELF", FALSE);
14     protocols[0] = xa_WM_SAVE_YOURSELF;
15     XmAddWMProtocols(toplevel, protocols, 1);

16     XmAddWMPProtocolCallback(toplevel, xa_WM_SAVE_YOURSELF,
17                             SaveYourselfCB, NULL);
18 }
```

lines 12-15 Include the **WM_SAVE_YOURSELF** atom in the **WM_PROTOCOLS** property on your application's top level window.

lines 16-17 Set up the callback to respond to the session manager's shut-down notification.

3. Prepare for shutdown when notified.

The **SetupSMProtocols(toplevel)** routine defined in the previous listing sets up the callback with which your application responds to **WM_SAVE_YOURSELF**. Include the following in your application's code to define the callback routine.

```

1 void
2 SaveYourselfCB(toplevel, client_data, call_data)
3 Widget          toplevel;
4 XtPointer        client_data;
5 XtPointer        call_data;
6 {
7     char          *cmd_argv[1];

8     cmd_argv[0] = GetBinaryFullPathName();

9     XSetCommand(XtDisplay(toplevel), XtWindow(toplevel), cmd_argv, 1);
10 }
```

lines 2-5 **toplevel** is your application's top level widget. **client_data** is callback client data. **call_data** is callback data. The latter two are not used in this example.

line 8 You should substitute the full pathname of your binary for **GetBinaryFullPathName()**. Include any command line options or special geometry information needed for the restart command.

line 9 Sets the **WM_COMMAND** property.

Because the session manager will be waiting for a **PropertyNotify** from the property **WM_COMMAND** as a confirmation that your application has saved its state, it is important that **WM_COMMAND** is updated even if its contents are correct. After updating **WM_COMMAND**, your application must wait to be shut down by the session manager. If your application receives a mouse or keyboard event after responding to **WM_SAVE_YOURSELF**, the shutdown has been canceled and your application may resume operation.

Chapter 12

Custom Distribution Mastering Toolkit

The Custom Distribution Mastering Toolkit (CDMT) allows you to prepare software distributions that conform to the Software Storage Object (SSO) standard. Use the CDMT to create distributions that can be installed with the **custom**(ADM) utility from cartridge tapes, floppy disks, CD, or directly from the hard disk over a network.

When the user installs the software product, **custom** transfers the files that comprise the product from the media (or directory on the hard disk) to the system on which the product is being installed. Using the information provided to CDMT, **custom** then places the files in the correct locations on the disk, assigns permissions, and executes configuration scripts.

To create SSO-standard products, use the CDMT to:

- Create simple custom-installable distributions (page 272)
- Create advanced software distributions (page 285), including:
 - Creating component scripts (page 285)
 - Creating software upgrades (page 298)
 - Using the SSO Path Mapping API (page 302)

Creating custom-installable software distributions

Here is an overview of the process of creating a simple **custom**-installable SSO-standard product with the CDMT:

1. Decide on a product structure (page 275)
Define your product hierarchy using the SSO standard.
2. Determine the attributes of each file (page 276)
Decide whether files are shared (read-only) or non-shared (read-write) and (optionally) public or private.
3. Set up the distribution tree (page 277)
Set up the source tree, check out the source code files from your source code version management system, build the source tree, and generate the distribution tree.
4. Set up your CDMT environment (page 278)
Before running the CDMT tools, you must set up a directory for the tools to generate their output files. Optionally, you can create a **cdmt.config**(CDMT) file to customize CDMT behavior.
5. Create the CDMT input files (page 278)
For each component of the product, create three CDMT input files:
 - Product description file (*.prd*)
 - Component description file (*.cmpnt*)
 - Package description file (*.pkg*)See **cdmtInput**(CDMT).
6. Create and populate the SSO tree (page 282)
Use **cdmtParse**(CDMT) and **cdmtCompress**(CDMT).
7. Create the **custom**-installable SSO archives (page 283)
Use **cdmtArchive**(CDMT).

For more advanced installations such as upgrades and installations that prompt the user for input, you can also create component scripts (page 285) that **custom** runs at installation time. Although the SSO standard does not require that you create component scripts, you can use these scripts to create more powerful and robust installations. See “Advanced software installations” (page 285).

See also:

- “About Software Storage Objects (SSO)” (this page)

About Software Storage Objects (SSO)

A Software Storage Object (SSO) is the fundamental building-block of a product. In the SCO system architecture, SSOs are the software “components” that comprise the product. Products consist of one or more components; components consist of one or more packages. See “About the new product structure” in the *Networking Guide* for more information.

A Software Storage Object:

- is a basic unit of software functionality and is always packaged and distributed as a whole.
- has a name and version number.
- contains private, internal data.
- has a well-defined external interface, defined by the developer.

A component is organized into a hierarchical tree of “packages”. A package is the smallest unit of software that can be installed or removed. Each package can contain other packages or files; the parent of each package is either another package or the top-level component. See “Product structure” (page 276).

The files in the component are grouped into a read-only software storage “object” in a directory subtree under the */opt* hierarchy, */opt/K/VendorCode/ComponentCode/ComponentVersion*. This directory is known as the *SSOroot* of the component. For example, the *SSOroot* of the WServer component is */opt/K/SCO/WServer/1.0.0*. For a description of the elements of *SSOroot*, see “Component identifiers” (page 274).

The CDMT tools arrange the distribution files into the SSO hierarchy and create **custom**-installable SSO archives. See “Creating and populating the SSO tree” (page 282) and “Creating custom-installable SSO archives” (page 283).

During the installation phases, **custom**(ADM) places the shared files in the *SSOroot* directory and copies the non-shared to the */var* hierarchy in client’s local file space */var/opt/K/VendorCode/ComponentCode/ComponentVersion*. See “Software management phases” (page 286).

See also:

- “Advantages of using SSOs” (page 274)

- “Software storage objects” in the *Networking Guide*
- “Product file locations” in the *Networking Guide*
- “Determining SSO file attributes” (page 276)
- **hierarchy(M)**

Component identifiers

The CDMT uses the following codes to identify elements of components:

<i>vendorCode</i>	A unique, trademarked string that identifies the vendor of the product; for example, SCO.
<i>componentCode</i>	A string that identifies the component, unique within the vendor’s namespace; for example, WServer.
<i>versionCode</i>	The version number of the component; for example, 3.2.5.
<i>packageCode</i>	A string that identifies the package within the component; for example, KSH.
<i>componentID</i>	A string that identifies the vendor, component, and version, in the format <i>vendorCode:componentCode::version</i> ; for example, SCO:WServer::1.0.0.
<i>packageName</i>	A string that identifies the vendor, component, and package, in the format <i>vendorCode:componentCode:packageCode</i> ; for example, SCO:unixos:KSH.
<i>packageID</i>	A string that identifies the vendor, component, package, and version, in the format <i>vendorCode:componentCode:packageCode::Version</i> ; for example, SCO:unixos:KSH::3.2.5.

NOTE Because *vendorCode*, *componentCode*, and *version* are elements of the path to the root of the component’s SSO, shorter names will result in better symbolic link performance.

Advantages of using SSOs

The advantages of cutting your product using the SSO standard are:

- provides object-oriented software management, configurable with software management tools
- provides diskless, small-disk, dataless, network client filesystem
- allows for more powerful network administration
- allows for more efficient use of disk space, as one copy of the product on the hard disk lets the system act as a standalone system, installation server, and a diskless or dataless server.

- allows multiple versions of a component to be loaded on the system and actually able to run on the system
- allows rollback of versions
- allows for simplified upgrades

Planning your product structure

The first step to creating a software distribution with CDMT is to decide on a product structure and organize your files in that structure:

- Divide your product into separate *components*; a component is a basic unit of functionality.
- Divide each component into *packages*; a package is a unit of software that can be installed or removed.
- Divide each package into *subpackages*, if appropriate.

See “Product structure” (page 276).

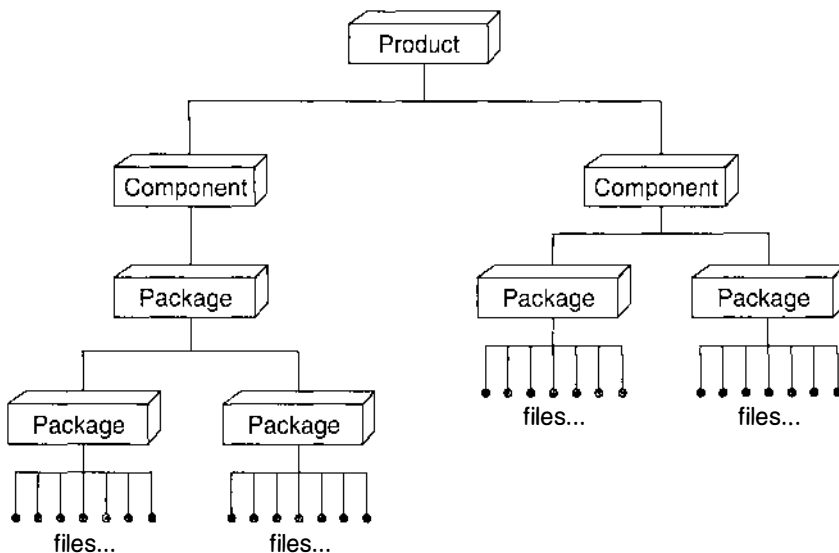
For example, the SCO OpenServer product is divided into components, such as **XClients** and **WServer**. Each of these components is divided into packages, such as **man**.

See also:

- “About the new product structure” in the *Networking Guide*

Product structure

The following graphic illustrates the structure of a product.



Determining SSO file attributes

Every file in an SSO has two attributes:

- | | |
|----------------------|---|
| shared or non-shared | A shared file is one that clients read, but do not modify; shared files remain read-only in the <i>/opt</i> hierarchy. A non-shared file is one that clients can modify; non-shared files are copied to the <i>/var/opt</i> hierarchy. |
| public or private | A public file is visible and accessible outside of the SSO; public files are linked to public directories (such as <i>/usr/bin</i>) so that components and users can access them. A private file is not accessible to other components and users. For example, back-end binaries, data files, and internal configuration files can be private files. |

In practice, an SSO generally consists of approximately 90% shared (read-only) data and 10% non-shared (read-write) data. As the developer, you must determine which files are shared and which files are non-shared.

To specify the shared and non-shared files when you create the distribution tree (page 272), put the shared files in the *SHARED* directory, and the non-shared files in *CLIENT*. Another way to accomplish this is to define the shared

and non-shared files in the CDMT input files. Set the **distTreeRootSHARED** attribute to the directory containing the shared files and **distTreeRootCLIENT** to the directory containing the non-shared files. Set these attributes in the **COMP** record in **.cmpnt*, or **PKG** record in **.pkg*. See **cdmtInput(CDMT)** for details.

For example, if you want **custom(ADM)** to place a non-shared file in the */var/opt/K/SCO/WServer/1.0.0/lib* directory, either put the distribution file in the **distTreeRoot/CLIENT/lib** directory or set the **COMP** record **distTreeRootCLIENT** attribute to the directory containing the non-shared files.

When you first convert to the SSO standard, mark all files in the SSO as public; later, you can specify private files using the SSO Path Mapping API (page 301).

To mark files as public, set the **FILE** record **exportPath** attribute in the **.pkg* file. See **cdmtInput(CDMT)**.

Setting up the distribution tree

Once you decide on the structure of your product, for each component in the product, generate a distribution tree using your standard development tools:

1. Set up your source tree, using the product structure you defined.

Get the files from your source code version management system (such as SCCS).

2. Build the source tree.

For example, run **make**.

3. Create the distribution (*dist*) tree. The *dist* tree contains the files in your component ready for distribution.

For example, run **make install**.

The root of the *dist* tree should be located relative to the top of the SSO for the component (*SSOroot*). The CDMT tools use the **distTreeRoot** attribute in the CDMT input files to determine the location of the distribution files. See **cdmtInput(CDMT)**.

Once you set up the distribution tree, you are ready to set up the CDMT environment.

Setting up the CDMT environment

To set up the CDMT environment:

1. Create the “working” directory for CDMT from which to read input and generate output. This should be any writable directory where you want the installable distribution to exist. For example, */build*.
2. Set the **CDMT_DIR** environment variable to the working directory. For Bourne or Korn shells (**sh** or **ksh**), enter:

```
CDMT_DIR=/build;export CDMT_DIR
```

3. Create the **\$CDMT_DIR/cdmt.config** file (optional).

The CDMT commands automatically read **\$CDMT_DIR/cdmt.config**, if it exists. Use this file to modify the default CDMT configuration. For example, you can specify the archive media and configuration for the final output. See **cdmt.config**(CDMT) for information on the format of this file.

By default, the CDMT tools use the directory specified by **CDMT_DIR**:

- CDMT tools read the CDMT input files in **\$CDMT_DIR/input**. See **cdmtInput**(CDMT).
- **cdmtParse**(CDMT) creates the **\$CDMT_DIR/sso** directory.
- **cdmtCompress**(CDMT) populates the **\$CDMT_DIR/sso** directory with SSOs to be archived.
- **cdmtArchive**(CDMT) uses the SSOs in **\$CDMT_DIR/sso** to generate and populate the **\$CDMT_DIR/archives** directory with archives that are **custom-installable** with the media type you specified.

Now, create the CDMT input files.

Creating CDMT input files

The CDMT input files define how the CDMT tools create your **custom-installable** software distribution.

For each product, you must create three types of CDMT input files:

- | | |
|---|---|
| Product description file (<i>.prd</i>) | contains information about the entire product and its components. |
| | In Example 12-1 (page 280), the <i>WServer.prd</i> file describes the product to which the WServer component belongs. |

Component description file (.cmpnt) contains information about the entire component and specifies the component structure, size of the component, and each package. Use this file to define component-wide default attributes for directories, files, links and symbolic links.

In Example 12-2 (page 280), the *WServer.cmpnt* file describes the WServer component.

Package description file (.pkg) contains information about packages and files in the component. Use this file to define information about each file in the distribution, such as owner, group, permissions, and exportPath. You can specify all the package descriptions in one file or create a separate file for each package.

NOTE The package description file corresponds to the file and package specification sections of the old SMT PERMLIST.

In Example 12-3 (page 281), the *WServer.pkg* file describes some of the packages in the WServer component.

For details on the format of these CDMT input files, see **cdmtInput(CDMT)**.

Create the CDMT input files and place them in **\$CDMT_DIR/input**. For example:

```
$CDMT_DIR/input/WServer.prd
$CDMT_DIR/input/WServer.cmpnt
$CDMT_DIR/input/WServer.pkg
```

You can also use the **cdmtConvert(CDMT)** tool to generate the CDMT input files from a list of relative pathnames output from **find(C)**.

Once you create the CDMT input files, you are ready to run the CDMT tools to create and populate the SSO tree. The following examples are product, component, and package description files for the WServer product.

Example 12-1 Example product description file (WServer.prd)

```

PROD:SCO:testProd:
    description      ="Widget Server"
    # $WServer_VER is a CDMT macro defined in the cdmtd.config file.
    version          =$WServer_VER
    packages         =SCO:WServer
    # $WServer_DIR is defined in cdmtd.config.
    cmpntFiles       =$WServer_DIR/input/WServer.cmpnt

```

Example 12-2 Example component description file (WServer.cmpnt)

```

COMP:SCO:WServer:
    description      = "Widget Server"
    # $WServer_VER is a CDMT macro defined in the cdmtd.config file.
    version          = $WServer_VER
    subpackages      = WSERVER,EXAMPLES
    # $WServer_DIR is defined in cdmtd.config.
    distTreeRoot     = $WServer_DIR
    # distTreeRootSHARED specifies where the shared files are located
    distTreeRootSHARED = $WServer_DIR
    # distTreeRootCLIENT specifies where the non-shared files are located
    distTreeRootCLIENT = $WServer_DIR
    pkgFiles         = $WServer_DIR/input/WServer.pkg

FILE_DEFAULT:
    perms            = 444
    owner            = bin
    group            = bin

DIR_DEFAULT:
    perms            = 755
    owner            = bin
    group            = bin

```

Example 12-3 Example package description file (WServer.pkg)

(This example describes only a few of the packages in the WServer component.)

```

PKG:Control:
    description      = "WServer component definition files"
    # $WServer_DIR is defined in cdmf.config.
    distTreeRoot     = $WServer_DIR
    # distTreeRootSHARED specifies where the shared files are located
    distTreeRootSHARED = $WServer_DIR
    # distTreeRootCLIENT specifies where the non-shared files are located
    distTreeRootCLIENT = $WServer_DIR

DIR:WSERVER:SHARED:./cntl:
    perms           = 0755
    owner           = bin
    group           = bin

FILE:WSERVER:SHARED:./cntl/ccs:
    perms           = 0755
    owner           = bin
    group           = bin

PKG:WSERVER:
    description      = "Widget Server"
    distTreeRoot     = $WServer_DIR
    distTreeRootSHARED = $WServer_DIR
    distTreeRootCLIENT = $WServer_DIR

DIR:WSERVER:SHARED:./bin:
    perms           = 0755
    owner           = bin
    group           = bin

FILE:WSERVER:SHARED:./bin/xm_wserver:
    perms           = 0755
    owner           = bin
    group           = bin
    exportPath       = /bin/xm_wserver

DIR:WSERVER:CLIENT:./usr:
    perms           = 0755
    owner           = bin
    group           = bin

DIR:WSERVER:CLIENT:./usr/lib:

```

```
perms          = 0755
owner          = bin
group          = bin

DIR:WSERVER:CLIENT:./usr/lib/X11:
perms          = 0755
owner          = bin
group          = bin

DIR:WSERVER:CLIENT:./usr/lib/X11/app-defaults:
perms          = 0755
owner          = bin
group          = bin

FILE:WSERVER:CLIENT:./usr/lib/X11/app-defaults/WServer:
perms          = 0444
owner          = bin
group          = bin
exportPath     = ./lib/X11/app-defaults/WServer
.
.
.
```

Creating and populating the SSO tree

Once the distribution tree and CDMT environment are ready, use the CDMT tools to create and populate the SSO tree:

1. Run **cdmtParse**(CDMT):

cdmtParse

The **cdmtParse** command:

- reads the **\$CDMT_DIR/cdmt.config** file, if it exists.
- reads the CDMT input files in **\$CDMT_DIR/input** (for example, *WServer.prd*, *WServer.cmpnt*, and *WServer.pkg*).
- creates the empty SSO tree prototype (*/opt* hierarchy) in the **\$CDMT_DIR/sso** directory.
- creates the SSO databases and places them in the SSO tree prototype.

2. Run **cdmtCompress**(CDMT):

cdmtCompress

The **cdmtCompress** command:

- reads the **\$CDMT_DIR/cdmt.config** file, if it exists.

- populates the `$CDMT_DIR/sso` directory with the component distribution files (from the distribution tree, specified with `distTreeRoot`). If specified, `cdmtCompress` compresses the distribution files. To specify compression, in the `cdmt.config` file, set the `compress` attribute in the `CONFIG` record to `TRUE`. See `cdmt.config(CDMT)`.
- runs `sum(C)` and determines the sizes of the distributions files and updates the SSO databases (in `$CDMT_DIR/sso`) with this information.
- runs `ssoPatch(CDMT)` on any specified files.

Now, create the **custom**-installable archives.

See also:

- Example 12-4, “CDMT make install script” (page 284)

Creating custom-installable SSO archives

After populating the SSO tree with the distribution files, run `cdmtArchive(CDMT)` to create the **custom**-installable archives:

`cdmtArchive`

The `cdmtArchive` command:

- reads the `$CDMT_DIR/cdmt.config` file, if it exists.
- reads the information from the `$CDMT_DIR/sso` directory.
- generates **custom**-installable archives and places the `cpio(C)` volumes in a subdirectory of `$CDMT_DIR/archives`.

By default, `cdmtArchive` creates cartridge tape `cpio` archives and places them in `$CDMT_DIR/archives/TAPE`. To change the default media type from `TAPE`, use the `-media` option:

`cdmtArchive -media media_type`

The `media_type` can be `TAPE`, `CD`, or `FLOPPY`.

To specify the default media in the `cdmt.config` file, use the `archiveMedia` attribute in the `CONFIG` record:

```
CONFIG:
archiveMedia    =      media_type
```

See `cdmt.config(CDMT)`.

Once you create the **custom**-installable archives, you can install them directly using **custom**(ADM) or copy them to a single media volume:

- To install the software directly to the hard disk from the **\$CDMT_DIR/archives/TAPE** directory:

```
custom -i -z $CDMT_DIR/archives/TAPE
```

- To copy the archives to floppy media, enter:

```
cdmtArchive -media FLOPPY -copy
```

This **cdmtArchive** command prompts you to insert a floppy and then copies the archives to the floppy disk. Use **custom** to test the installable media. See Chapter 4, “Installing and managing software components” in the *SCO OpenServer Handbook*.

Example 12-4 (this page) is a sample script that you can include in your **make install** process, that runs **cdmtParse**, **cdmtCompress**, and **cdmtArchive**.

See also:

- **dosfloppy**(CDMT)

Example 12-4 CDMT make install script

```
#!/bin/sh

CDMT_DIR=/build

cdmtParse
if test $? != 0
then
    exit 1
fi

cdmtCompress
if test $? != 0
then
    exit 1
fi

cdmtArchive
if test $? != 0
then
    exit 1
fi
```

Advanced software installations

The CDMT tools allow you to create simple **custom**-installable software distributions. However, to create more advanced installations such as upgrades and installations that prompt the user for input, you should also create component scripts (this page).

To create upgrades, see “Creating software upgrades” (page 298).

In addition, to take full advantage of the SSO standard, you can mark specific files in your distribution as private. To do this, use the SSO Path Mapping API (page 301).

About component scripts

“Component scripts” are scripts which function as an API between **custom**(ADM) and the component; **custom** runs these scripts during installation of the component.

The **custom** utility recognizes two types of component scripts:

- component control scripts (**ccs**)
- component query scripts (**cqs**)

As the component developer, use these component scripts to:

- control each phase of the installation
- control the removal process
- create product upgrades
- prompt the user for information
- use input from user

NOTE Creating **ccs** and **cqs** scripts is not mandatory to creating a **custom**-installable product with CDMT; use these scripts for more advanced installations and upgrades.

The syntax for the **ccs** is:

ccs *step* [*keyword_list*] *package_list*

See “Software management phases” (page 286) for information about the *step* option and **ccs**(CDMT) for information about the other arguments to **ccs**.

The syntax for the **cqs** is:

cqs [*keyword_list*] *package_list*

See **cqs**(CDMT) for information about the **cqs**.

See also:

- `ccsUpgradeTool(CDMT)`

Software management phases

The installation process for a component is divided into five “software management phases”. Using the information in the `ccs(CDMT)` and `cqs(CDMT)` scripts, `custom(ADM)` performs these phases:

LOAD	copies the component from the media to the hard disk in the <code>/opt</code> hierarchy. The files in <code>/opt</code> remain read-only.
ATTACH	copies the non-shared (per-client) files from the pristine versions in the SSO in the <code>/opt</code> hierarchy to the component’s private area in <code>/var/opt</code> . The files in <code>/var/opt</code> are read-write.
REGISTER	makes the configuration interfaces for components that provide “system services” ¹ available to components that “consume” system services.
CONFIGURE	configures components that consume system services into the system services.
EXPORT	makes the component’s public files in <code>/opt</code> visible by symbolic-linking them to external directories, such as <code>/usr/bin</code> .

Each phase consists of three steps:

pre step	performs component-specific preparation, defined by the developer
system step	<code>custom</code> performs specified operations in the <code>ccs</code> script
post step	performs component-specific clean-up operations, defined by the developer

The component developer controls the **pre** and **post** steps using the component control script (`ccs`).

NOTE The `ccs` is not required to take action during each step. If it does not need to perform any work during a step, the `ccs` should exit with a value of 0.

The removal process runs backward through the phases, prepending the prefix **UN** to the step name.

1. For example, the link kit and `netconfig` are system services. In general, SCO components are system service providers, other applications are system service consumers.

The **ccs** phases look like this:

Installation	Removal
ccs PRE_LOAD load system step ccs POST_LOAD	ccs PRE_UNEXPORT unexport system step ccs POST_UNEXPORT
ccs PRE_ATTACH attach system step ccs POST_ATTACH	ccs PRE_UNCONFIGURE unconfigure system step ccs POST_UNCONFIGURE
ccs PRE_REGISTER register system step ccs POST_REGISTER	ccs PRE_UNREGISTER unregister system step ccs POST_UNREGISTER
ccs PRE_CONFIGURE configure system step ccs POST_CONFIGURE	ccs PRE_UNATTACH unattach system step ccs POST_UNATTACH
ccs PRE_EXPORT export system step ccs POST_EXPORT	ccs PRE_UNLOAD unload system step ccs POST_UNLOAD

See also:

- “LOAD and UNLOAD phases” (this page)
- “ATTACH and UNATTACH phases” (page 288)
- “REGISTER and UNREGISTER phases” (page 289)
- “CONFIGURE and UNCONFIGURE phases” (page 290)
- “EXPORT and UNEXPORT phases” (page 290)
- **ccs**(CDMT)

LOAD and UNLOAD phases

During the **LOAD** phase, **custom**(ADM) transfers all the selected packages from the source media to the target media, in preparation for installing the local system, diskless clients of the local system, or remote systems across the network. **LOAD** creates the */opt* portion of the SSO.

NOTE You can legally load a component onto a system on which the component cannot run; for example, the administrator might want to load the component on a server and **ATTACH** it to network clients without attaching it to the server.

- pre step** **ccs PRE_LOAD keyword list package list**
 Only the files in the **SharedControl** package of the component are loaded on the target.
- system step** If disk space is at a premium, **custom** might execute the remove procedure for the old component. **custom** moves the files from the distribution media to the SSO on the hard disk.
- post step** **ccs POST_LOAD keyword list package list**

During the **UNLOAD** phase, **custom**(ADM) removes the */opt* portion of the component's SSO.

- pre step** **ccs PRE_UNLOAD keyword list package list**
- system step** **custom** removes */opt*.
- post step** **ccs POST_UNLOAD keyword list package list**
 The only files that remain after a full removal are the files in the component's control package (the files in the *SSOroot/cntl* directory).

ATTACH and UNATTACH phases

During the **ATTACH** phase, **custom** creates and initializes the per-client portion of the SSO, the */var/opt* portion of the SSO. **ATTACH** copies the non-shared files and establishes the client contents database. The software management tools use the database to track, checksum, and perform other per-client file management.

The **ATTACH** phase is always executed once per client, whether the client is the local system, a diskless client, or a dataless client.

Once the **ATTACH** phase is complete, applications which do not require configuration into a system service (such as the link kit), should be capable of running.

NOTE **custom** sets the **PATH** variable to the */ibin* directory. Any search invocations outside */ibin* or the SSO itself are not possible until the external links are created during the **EXPORT** phase.

However, the **ccs** should perform all internal configuration (configuration that can be accomplished by manipulating files within the SSO or, in the case of an upgrade, using configuration parameters from the previous version.

- pre step** **ccs PRE_ATTACH keyword list package list**
 The **ccs** checks for version compatibility. If it is not valid to install the component on the client, the **ccs** should return an exit code of 1 (**FAIL**).

system step The software management tools copy the non-shared files into the */var/opt* area of the SSO.

post step **ccs POST_ATTACH keyword list package list**

If either the **OLD_CUSTOM_UPGRADE** or **UPGRADE** keyword is specified, the **ccs** should merge old configuration data from the **\$CCS_PERSISTENT_STORAGE** directory into the new files.

Once the data has been restored, the **ccs** should perform any specified configuration using parameters gathered from the **cqs**. Only files in the */var/opt* area can be modified.

During the **UNATTACH** phase, the **ccs** removes the */var/opt* portion of the SSO.

pre step **ccs PRE_UNATTACH keyword list package list**

system step The **ccs** removes the */var/opt* portion of the SSO. This action removes any specific component configuration and leaves the client as if the component had never been installed.

post step **ccs POST_UNATTACH keyword list package list**

REGISTER and UNREGISTER phases

For components that provide system services, the **ccs** makes the component's system services available to other components during the **REGISTER** phase.

pre step **ccs PRE_REGISTER keyword list package list**

system step **custom** performs an "early export" of files or interfaces that are involved in providing a system service. These files are marked in the CDMT input files by the **exportPathRegPhase** attribute. See **cdmtInput(CDMT)**.

post step **ccs POST_REGISTER keyword list package list**

For components that provide system services, the **ccs** removes those services during the **UNREGISTER** phase.

pre step **ccs PRE_UNREGISTER keyword list package list**

The **ccs** performs any actions necessary prior to the removal of system services. Depending on the service, the **ccs** might inform all service consumers of the removal of the service.

- system step** **custom** removes filenames that were “exported early” during the **REGISTER** phase. No changes are made to either the */opt* or */var/opt* area of the SSO.
- post step** **ccs POST_UNREGISTER** *keyword list package list*

CONFIGURE and UNCONFIGURE phases

During the **CONFIGURE** phase, the **ccs** configures the component into any system services that it consumes. For example, the component might add a driver to the link kit using the **idinstall**(ADM) utility.

- pre step** **ccs PRE_CONFIGURE** *keyword list package list*
- system step** No action is defined.
- post step** **ccs POST_CONFIGURE** *keyword list package list*
- The **ccs** performs any remaining configuration that relied upon actions taken in the **PRE_CONFIGURE** and system step.

During the **UNCONFIGURE** phase, the **ccs** removes the component from any system services that it consumes. For example, remove kernel device drivers using a call to **idinstall**(ADM).

- pre step** **ccs PRE_UNCONFIGURE** *keyword list package list*
- system step** No action is defined.
- post step** **ccs POST_UNCONFIGURE** *keyword list package list*

EXPORT and UNEXPORT phases

During the **EXPORT** phase, the **ccs** exports all remaining public interfaces to the public space.

- pre step** **ccs PRE_EXPORT** *keyword list package list*
- system step** The file name space is exported to directories external to the SSO.
- post step** **ccs POST_EXPORT** *keyword list package list*

During the **UNEXPORT** phase, the **ccs** removes files that were exported in the **EXPORT** phase.

pre step	ccs PRE_UNEXPORT <i>keyword list package list</i>
system step	The exported filenames are removed. No changes are made to either the <i>/opt</i> or <i>/var/opt</i> areas of the SSO.
post step	ccs POST_UNEXPORT <i>keyword list package list</i>

Component script guidelines

These general guidelines apply to both **ccs**(CDMT) and **cqs**(CDMT) component scripts:

Writing component scripts

- If you want to use the standard library, **ccsSetup**(CDMT), write your component scripts in **sh**(C) or **ksh**(C) and include the standard library */libin/ccsSetup.sh*.
- Place component scripts in the *SSOroot/cntl* directory.

SSO pathnames A component can find the */opt* and */var/opt* pathnames to its own SSO by examining these environment variables:

- **SSO_SHARED_ROOT**
- **SSO_CLIENT_ROOT**
- **SSO_EXPORT_TARGET**

See “Component script environment variables” (page 296) for details.

Communicating between scripts

Use the **CCS_PERSISTENT_STORAGE** environment variable to specify a directory that different phases of the **ccs** can use to communicate with other phases across invocations of the **ccs**. See “Component script environment variables” (page 296).

Control packages Until the **LOAD** phase, **custom**(ADM) has only installed the target with the files included in the **SharedControl** package. All files in the **SharedControl** package must be located in the *SSOroot/cntl* directory.

Because the **cqs** might run before the **LOAD** phase, it must only refer to files in **SharedControl**. The **ccs** can only refer to files in **SharedControl** until the **PRE_LOAD** step.

List both the **ccs** and **cqs**, along with any package-level subscripts, in the **SharedControl** package.

Utilities available to component scripts

The **ccs** and **cqs** must only reference scripts and executables that are located in either:

- the component itself

In this case, refer to the file by the full SSO pathname. Pass the pathname to the root of the SSO to the **ccs** using the **SSO_SHARED_ROOT** environment variable. See “Component script environment variables” (page 296).

- the */ibin* directory

custom sets the **PATH** variable to */ibin*. The **ccs** and **cqs** scripts should *not* use the full pathname to refer to files in */ibin*. For example, use *sed*, not */ibin/sed*.

After the **POST_CONFIGURE** step, the **ccs** can call other executables directly.

Trapping interrupts

Component scripts can trap interrupts, but should allow the user to interrupt the script. For example, a trap might call a cleanup function to remove temporary files. To trap interrupts during a critical, non-interruptible phase, include a trap that does nothing in the script; immediately after the critical section, reset the trap.

Relinking the kernel

To schedule kernel relinks in the **ccs**, use the **customSched**(CDMT) utility.

Determining what is currently installed

To determine which components are installed on the system, use the **customquery**(ADM) utility.

See also:

- “Component script structure” (page 293)
- “Software management phases” (page 286)

Component script structure

As the developer, you control the structure of the **ccs** and **cqs** scripts. However, for consistency, we suggest this general structure for your scripts:

1. Global component initialization actions
2. Loop to call each package script
3. Global component cleanup actions

Create separate package scripts for each package-specific action. Package scripts should be passed the same argument list and should return the same range of exit values as its parent **ccs** or **cqs**. Place package-specific **ccs** and **cqs** scripts in the subdirectory specified by *packageCode* of the *SSOroot* directory.

NOTE Before the top-level **ccs** or **cqs** creates the package directory, it must convert the fully-qualified package name to the short name. Use **getPackageCode** in **ccsSetup**(CDMT).

This structure makes it easier to treat packages as modular units that can be added to components.

For an example **ccs** script, see Example 12-5 (page 294).

See also:

- “Component script guidelines” (page 291)
- **ccs**(CDMT)
- **cqs**(CDMT)
- **ccsSetup**(CDMT)

Example 12-5 Example ccs script

```

#!/bin/sh
#*****
#
#                               XClients ccs script
#-----
# Top-level component control script (ccs) for the XClients component.
#-----
#   input:
#       $1 is the name of the step
#       $2 is the keyword list (e.g., UPGRADE )
#       $3 is the package list
#=====

# Read arguments into variables
scriptname="$0"
step="$1"
keywords="$2"
pkglist="$3"

# Source in the standard functions library, ccsSetup.sh

# Set global variables indicating whether this is an upgrade or removal.

sso_upgrade=
old_upgrade=
for word in $keywords
do
    case "$word" in
        UPGRADE) sso_upgrade=TRUE
                ;;
        OLD_CUSTOM_UPGRADE) old_upgrade=TRUE
                ;;
    esac
done

# Set the appropriate NLSPATH to enable the "dspmsg" utility to work later
# in this script. Note that pkg scripts should not have to worry about
# NLSPATH, as they should inherit the copy set by the top-level ccs script.

NLSPATH="${SSO_SHARED_ROOT}/cntl/msg/%L/%N.cat":"${NLSPATH}"
export NLSPATH

#===== INT ===
# doPostattach --
#

```

```

# In the case of an upgrade of an old component, doPostattach merges the
# saved files (in CCS_PERSISTENT_STORAGE) into the new
# configuration files (located under SSO_CLIENT_ROOT).
#-----
doPostattach()
{
    # if it's not an upgrade, we're done.
    [ "$upgrade" ] || return
}

#===== INT =====
# doPostconfig --
#
# doPostconfig performs inter-component tasks, such as setting up
# cron jobs, desktop customizations, and installing drivers into
# the link kit.
#-----
doPostconfig()
{
    # ... change the cron file using configuration information
    #     supplied in installation

    # ... install cron file with crontab

    # ... schedule a kernel relink using customSched(CDMT)
    :
}

#===== INT =====
# main --
#
# Based on step specified, perform actions of the general form:
#     Global initialization
#     Loop to call each package script for package-specific actions
#     Global cleanup
#-----

# the exit value from this script
ccs_return_value=0

# These fns get called prior to running package-specific scripts.
case "$step" in
    POST_ATTACH)      doPostattach
                      ;;
    POST_CONFIGURE)    doPostconfig
                      ;;
esac

# the directory containing whatever package scripts we've got
pkgscripts=${SSO_SHARED_ROOT}/cntl/packages

```

```

# For each package, run that package's package script with
# the same arguments as this script got.
# Record any exit values from the package scripts.
for pkg in $pkglist
do
    short_pkg_name=`getPackageCode $pkg`
    [ -f ${pkgscripts}/${short_pkg_name}/ccs ] && {
        ${pkgscripts}/${short_pkg_name}/ccs $step $keywords "$pkglist"

        # This script exits with the most severe of exit code of any of
        # its package scripts. An exit of 1 means "fail", which is more
        # severe than an exit of 2, which means "warning". An exit of 0
        # (success) is the least severe.
        case "$?" in
            1) ccs_return_value=1
               ;;
            2) [ "$ccs_return_value" -eq 0 ] && ccs_return_value=2
        esac
    }
done

exit $ccs_return_value

```

Component script environment variables

The following environment variables are available for use in **ccs** and **cqs** scripts:

SSO_SHARED_ROOT	absolute pathname to the root of the <i>/opt</i> area for the component; available to both ccs and cqs scripts.
SSO_CLIENT_ROOT	absolute pathname to the root of the <i>/var/opt</i> area for the component; available only to ccs scripts in all phases after LOAD .
SSO_EXPORT_TARGET	target directory of an export (<i>/usr</i> , unless otherwise specified); available only to ccs scripts in the EXPORT phase.
CCS_PERSISTENT_STORAGE	absolute pathname to a directory that is guaranteed to exist across invocations of the ccs , once the LOAD phase completes; available to both ccs and cqs scripts. For example, use this directory to pass parameters from the cqs to the ccs .

Note the following restrictions when using **CCS_PERSISTENT_STORAGE**:

- This variable is component- and version-specific, thus you cannot use it to pass parameters between components.
- Do not use this variable to store permanent data for administration utilities of the component outside the software management operation. Use the */var/opt* area of the SSO instead.
- Do not use the directory specified by **CCS_PERSISTENT_STORAGE** to store data for use between an installation and a remove operation. Use the */var/opt* area of the SSO instead.
- The component must create the **CCS_PERSISTENT_STORAGE** directory structure because **custom(ADM)** does not impose any usage policy on this directory.
- The actual value of this variable might change between software management phases, so components should not expect this directory to be located in the same place throughout the installation. However, the files in the directory remain untouched and are copied without changes.

SSO_UPGRADE_COMPONENTS

list of existing components to be upgraded; available to both **cqs** and all phases of the **ccs** scripts. **SSO_UPGRADE_COMPONENTS** lists only components to be upgraded, not packages.

See **ccsSetup**(CDMT) for the variables set in **ccsSetup** library.

The following keywords are passed to the **ccs** and **cqs** scripts:

OLD_CUSTOM_UPGRADE keyword that specifies that this is an upgrade from an old (*perms-list*) component to a new (SSO-based) component; available to the **cqs** and all phases of the **ccs** scripts. The **cqs** should not run interactively when it receives the **OLD_CUSTOM_UPGRADE** keyword. Instead, **cqs** should save any configuration information to the **CCS_PERSISTENT_STORAGE** directory.

UPGRADE

keyword that specifies that this is an upgrade from an old version of an SSO-based component to a new version of an SSO-based component; available to the **cqs** and all phases of the **ccs** scripts.

See also:

- **ccs**(CDMT)
- **cqs**(CDMT)

Creating software upgrades

An “upgrade” is an installation that preserves configuration information from a previously installed version of the component and allows the component to use that configuration.

During an upgrade, **custom**(ADM) must determine the following information for each component:

- Which *components* currently on the system (if any) to upgrade to the new version.
- Which *packages* (if any) of the new component to install.

To determine which existing components to upgrade, **custom** reads the following attributes in the CDMT input files:

upgradeVersions defines the component versions that the component can upgrade. Use this attribute when the new component is a new version of an older component with the same name. See “Specifying the upgrade versions” (page 299).

upgradeMapping defines the component or package names that the component can upgrade. Use this attribute if the component or package name or location changed between versions. See “Specifying packages to upgrade” (page 300).

See **cdmtInput**(CDMT) for more information about these attributes.

If **custom** detects an upgradable version on the system being installed (the components specified by **upgradeVersions** and **upgradeMapping** exist on the system), it passes the **SSO_UPGRADE_COMPONENTS** environment variable to the **ccs** and **cqs** scripts. This variable defines the list of existing components to upgrade. If the **SSO_UPGRADE_COMPONENTS** list contains at least one item, the component will be installed in “upgrade mode”.

In upgrade mode, **custom** passes either the **UPGRADE** or **OLD_CUSTOM_UPGRADE** keyword to the **ccs** and **cqs** scripts (depending on whether the component to upgrade was cut to the SSO standard using the CDMT or to the old *perms-list* standard using the Software Mastering Toolkit (SMT)).

NOTE When the **cqs** receives either the **UPGRADE** or **OLD_CUSTOM_UPGRADE** keywords, it should run non-interactively and save any old configuration information in the directory specified by the **CCS_PERSISTENT_STORAGE** variable. See “Component script environment variables” (page 296) for more information.

To determine which packages of the new component to install, **custom** uses command-line options. See **custom(ADM)** for details.

See also:

- **ccs(CDMT)**
- **cqs(CDMT)**
- **ccsUpgradeTool(CDMT)**

Specifying the upgrade versions

To specify the previous versions of the component to upgrade, use the **upgradeVersions** attribute.

Set **upgradeVersions** in the **COMP** record in the *.cmpnt* file:

```
COMP:vendorCode:componentCode:
upgradeVersions = versionExpr
```

where *versionExpr* is a regular expression. For information about regular expressions, see **regexp(TCL)**.

For example, to specify an upgrade of the WServer component from either 4.0.0a or 4.2.0a to version 5.0.0a, use these record attributes in the *.cmpnt* file:

```
COMP:SCO:WServer:
version          = 5.0.0a
upgradeVersions = ^4\\.0\\.0a$
upgradeVersions = ^4\\.2\\.0a$
```

NOTE The string “*^4\\.2\\.0a\$*” is a regular expression which matches the version “4.2.0a” and nothing else.

Another way to write the two **upgradeVersions** lines is:

```
upgradeVersions = ^4\\.([02])\\.0a$
```


If more than one version of the component installed on the system matches the **upgradeVersions** attribute, **custom**(ADM) upgrades the latest version only. In the example, if both version **COMP:SCO:WServer:4.0.0a** and version **COMP:SCO:WServer:4.2.0a** are installed on the system, **custom** upgrades only version 4.2.0a to version 5.0.0a.

If the name of a component changes between versions or a package changes from one component to another, you must use the **upgradeMapping** attribute in the **COMP** record. See “Specifying packages to upgrade” (this page).

See also:

- **cdmtInput**(CDMT)

Specifying packages to upgrade

To specify the packages to upgrade to the new version, use the **upgradeMapping** attribute. This attribute supplements **upgradeVersions** by providing the ability to upgrade an old component or package name to the new name. Use **upgradeMapping** in cases where functionality has moved from one component to another between releases or when the names of the packages in a component change.

To use **upgradeMapping**:

- Set **upgradeMapping** in the **COMP** record in the *.cmpnt* file:

```
COMP:vendorCode:componentCode:
upgradeMappings = vendorCode:componentCode::versionExpr
```

where *versionExpr* is a regular expression. For information about regular expressions, see **regexp**(TCL). For example:

```
COMP:SCO:WServer:
upgradeMappings = SCO:WServer::^4\\.0\\.0a$
```

- Set **upgradeMapping** in the **PKG** record in the *.pkg* file:

```
PKG:packageCode:
upgradeMappings = vendorCode:componentCode:packageCode::versionExpr
```

For example:

```
PKG:WServer:
upgradeMappings = SCO:WidgetServer:MAN::^3\\.0\\.0$
```

See also:

- “Specifying the upgrade versions” (page 299)
- **cdmtInput**(CDMT)

About the SSO Path Mapping API

The SSO Path Mapping API is a utility function that enables you to construct an SSO so that only a few files are “public” (exported to the public space) and the majority of the files are “private” (visible only to programs in the component).

Marking files that are only accessed directly by programs in the component as private reduces the possibility of:

- name space conflicts between unrelated components
- accidental dependencies between components

See “About Software Storage Objects (SSO)” (page 273) for information about private and public files.

When you mark a file as private, it exists only in the SSO and is not exported to a public directory. Because private files are not exported, even files within the SSO cannot locate them unless you place hardcoded references to the private files in your source. However, in this case, if the version or name of the component changes, you must change the references and recompile the programs. Instead of placing hardcoded references in your source, use the SSO Path Mapping API.

The API cooperates with **custom**(ADM) to allow the component to find the location of its own private files, so that the component does not need to make all of its files public.

The Path Mapping API adds a patch area to scripts or binaries. When you recut the component, the CDMT tools run **ssoPatch**(CDMT); this utility dynamically changes this patch area whenever the path to the root of the SSO (**SSOroot**) changes. For example, in an update, the original component is updated with new files to create a new version. In this case, **custom** creates the SSO for the new version at installation time, adding files from the original SSO to the update distribution, and patching those files with the new SSO’s version number.

See also:

- “Using the SSO Path Mapping API” (page 302)
- **cdmtInput**(CDMT)

Using the SSO Path Mapping API

To specify that the CDMT tools run the **ssoPatch** utility on a file when you recut the component:

1. Set the **flags** attribute to **ssoPatch**:
 - in the **.cmpnt* file, set **flags** in the **FILE_DEFAULT** record.
 - in the **.pkg* file, set **flags** in the **FILE** record.

See **cdmtInput(CDMT)**.

2.

C and C++ programs link the API, */usr/lib/libssso.a*, into the source file.
(page 302)

Tclscripts (page 304) include a patch area definition in the script.

Shell scripts (page 304) run the **ssoPathMap(CDMT)** utility in the script.

C and C++ programs API

The CDMT includes two files that enable you to use the SSO Path Mapping API with C and C++ programs:

/usr/include/sso/pathMap.h Include this file in the source.

/usr/lib/libssso.a Link this file in when you compile the source.

For example:

cc options -lssso

```
#include <sso/pathMap.h>
```

```
SSO_ALLOC_ID_OBJECT(uniqueName, "vendorCode", "componentCode", "version");
```

```
const char * SSO_GetSegmentRootPath(options, ssoIdObject);
unsigned options;
ssoIdObject_t *ssoIdObject;
```

```
char * SSO_GetAbsolutePath(options, ssoIdObject, relPath);
unsigned options;
ssoIdObject_t *ssoIdObject;
const char *relPath;
```

SSO_ALLOC_ID_OBJECT This macro creates a “patch area” in the binary; the software management utilities can change this area dynamically. The macro creates the variable *ssoIdObject_t uniqueName*, initialized

to the values *vendorCode*, *componentCode*, and *version*. These arguments are literal names; the macro will convert them to C strings. You can override the values later using the **ssoPatch**(CDMT) utility.

SSO_GetSegmentRootPath This function returns the pathname of the root directory of the SSO (*SSOroot*). This is determined by analyzing the patch area *ssoIdObject*, which should be created by a previous call to **SSO_ALLOC_ID_OBJECT**. The string returned by **SSO_GetSegmentRootPath** is static memory and should not be deallocated using **free**(S).

The *options* argument is a bit flag constructed from one or more of:

SSO_SHARED the function returns the path to the shared read-only SSO prototype (the */opt* portion of the SSO).

SSO_CLIENT the function returns the path to the client-specific read-write SSO location (the */var/opt* portion of the SSO).

SSO_NO_ABORT if the function is unable to return the requested information, it returns **NULL** instead of aborting. If **SSO_NO_ABORT** is not specified, the function aborts the execution.

SSO_GetAbsolutePath This function returns the absolute pathname of the SSO internal file named by *relPath*. *relPath* specifies the pathname of an internal SSO file, relative to the root of the SSO.

The *options* and *ssoIdObject* arguments are the same as with **SSO_ALLOC_ID_OBJECT**.

The memory returned by the function **SSO_GetAbsolutePath** is dynamically allocated and should be deallocated with a call to **free** when it is no longer needed.

If you set the **SSO_NO_ABORT** flag, the **SSO_GetAbsolutePath** and **SSO_GetSegmentRootPath** functions can return the following values:

EINVAL Patch area in the *ssoIdObject* is not valid.

ERANGE Specified *options* flag is illegal.

ENOMEM Insufficient memory is available.

In cases where a static library from one component is intended to be linked with programs from another component, it is not possible for the static library to use the Path Mapping API. The actual text of the static library is located in the SSO of the executable, not the SSO of the library, so **custom** does not patch the library if the path to the SSO of the library changes.

Tcl scripts API

The SSO Path Mapping API functionality is built into the Tcl interpreter.

To use path mapping with Tcl scripts, include a patch area definition in the script. For example:

```
# Patch area definition
```

```
set lclPatchArea {_SSO_ID_ST@vendorCode:componentCode:version @SSO_ID_END_}
```

```
proc SSO_GetSegmentRootPath [SHARED | CLIENT] $lclPatchArea
```

The patch area definition must appear in the actual script being patched (not in a library). The string between the “@” symbols can be any length, but in practice should be no shorter than 35 characters. Set *vendorCode*, *componentCode*, and *version* to initial defaults, similar to the fields in the **SSO_ALLOC_ID_OBJECT** macro in the “C and C++ programs API” (page 302).

The Tcl **SSO_GetSegmentRootPath** function returns an absolute path to the root of the SSO (*SSOroot*). Use the **SHARED** keyword for the read-only (*/opt*) SSO and **CLIENT** for the read-write (*/var/opt*) SSO.

Bourne and Korn shell scripts API

The Bourne and Korn shell script API is similar to the “Tcl scripts API” (this page), except that it depends upon the **ssoPathMap**(CDMT) utility program, instead of a built-in shell function.

Include a patch area definition in the shell script using the API. For example:

```
# patch area definition
```

```
id="_SSO_ID_ST@vendorCode:componentCode:version @SSO_ID_END_"
```

```
ssoPathMap [-f relPath] [ -s segmentId] -i ssoIdObject
```

The patch area definition must appear in the actual script being patched (not in a library). The string between the “@” symbols can be any length, but in practice should be no shorter than 35 characters. Set *vendorCode*, *componentCode*, and *version* to initial defaults, similar to the fields in the **SSO_ALLOC_ID_OBJECT** macro in the “C and C++ programs API” (page 302).

The **ssoPathMap** utility is used to interpret the patch area. If you omit the *-f relPath* argument, **ssoPathMap** returns the path to the root of the SSO (*SSOroot*).

See also:

- **ssoPathMap**(CDMT)

Bibliography

The following commercial books and articles provide details about topics overviewed in the SCO OpenServer Development System. This list is provided for reference only; it is not comprehensive and The Santa Cruz Operation, Inc. does not guarantee the accuracy of these publications. This list is divided into two parts: books and periodicals.

Books

- 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.

More Programming Pearls: Confessions of a Coder. Reading, MA: Addison-Wesley, 1988.

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.
- Dowd, Kevin. *High Performance Computing*. Sebastopol, CA: O'Reilly & Associates, 1993. Discusses the basics of modern workstation architecture, understanding and writing performance benchmarks, and methods for improving performance of critical applications.
- Ellis, M. and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley, 1990.
- Feit, Sidnie. *TCP/IP: Architecture, Protocols, and Implementation*. New York, NY: McGraw-Hill, 1992. Explains how to implement a TCP/IP network. Includes detailed instructions, many examples, and reference information. Covers core protocols plus additional services and products: NFS, NIS, BIND, ARP, RIP, KERBEROS, SNMP, program interfaces. How to invoke network services, plan your name/address structure, troubleshoot network problems, connect with bridges and routers.
- Gauthier, R. *Using the UNIX System*. Reston, VA: Reston Publishing Company, 1981.
- Gircys, Gintaras R. *Understanding and Using COFF*. (Nutmash 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.
- Intel
Intel486™ Microprocessor *Hardware Reference Manual*. Describes the 486 architecture for use by hardware designers and system programmers.

Intel486™ Microprocessor Family, *Programmer's Reference Manual*. 1992. Provides information about the 486 architecture for applications and system programmers. Includes information about the instruction set, the FPU, and compatibility with other Intel microprocessors.
- Kernighan, Brian W. and P.J. Plauger.
Software Tools. Reading, MA: Addison-Wesley, 1976.

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. This is one of the classic books about the programming philosophy for UNIX operating systems, although it is now somewhat dated. When it was published, there was only one shell available, networking and graphical user interfaces were not an issue, and many other features that have become standard were not yet included.

Kettle, Peter and Steve Statler,

Writing Device Drivers for SCO UNIX: A Practical Approach. Addison-Wesley, Summer 1992. Discusses how to write device drivers for UNIX System V version 4, including carefully graded exercises for writing a simple line discipline a real STREAMS driver, and other commonly-requested driver problems. The book includes answers to the exercises that are targeted explicitly to the Version 4 product, as well as background information about the kernel in the Intel 80x86 processors.

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.

Levine, John.

lex & yacc. Sebastopol, CA: O'Reilly & Associates, Second edition, 1992. Shows programmers how to use the `lex`(CP) and `yacc`(CP) utilities in program development. The second edition contains revised tutorial sections for novice users and reference sections for advanced users.

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.

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.
- Pajari, George. *Writing UNIX Device Drivers*. Addison-Wesley, 1991. Provides details and a hands-on, nuts and bolts approach to writing drivers for SCO operating systems. The book includes a sample SCSI disk driver, information about intelligent serial I/O adapters, STREAMS for token ring, and so forth. A diskette of sample drivers and exercises is available separately.
- Plum, Thomas, and Jim Brodie.
Efficient C. Cardiff, NJ: Prentice-Hall, 1985.
Learning to Program in C. Cardiff, NJ: Prentice-Hall, 1983. Provides an introduction to (or refresher on) the C programming language.
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.
- Rao, Bindu R. *C++ and the OOP Paradigm*. New York, NY: McGraw-Hill, 1992. Introduces C++ and object oriented programming with examples in C++.
- 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. This book is especially useful if you are dealing with XENIX programs; it contains information about features such as XENIX semaphores and XENIX file locking. The Stevens and Lewine books listed in this bibliography are more up-to-date general UNIX programming guides.
- Rosenberry, Ward, David Kenney, and Gerry Fisher,
Understanding DCE. Sebastopol, CA: O'Reilly & Associates, 1992. A technical and conceptual overview of OSF's Distributed Computing Environment (DCE) for programmers and technical managers, marketing, and sales people.

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**(CP).

SCO

Directory. A listing of add-on software and hardware products available for SCO systems.

Programming Tools Guide. Provides generally useful information about the programming and debugging tools and their use in developing and maintaining software, as well as tracking down and eradicating problems in C and assembly language programs. The C compiler, its use in practice, explanations of compiler error messages, and implementation-defined behavior of the ANSI compiler are all presented. In addition, the book includes information about using **dbXtra**(CP) to debug C and C++ programs in a Motif-based windowing environment. Shipped as part of the Development System documentation set.

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 SCO features added to the porting base.

Programmer's Reference Manual. A set that includes two volumes of manual pages for the UNIX Development System, including sections (CDMT), (CP), (FP), and (S), a volume for the X Window libraries, one volume for the Motif library, and manual pages for special networking libraries. The printed books are shipped as part of the Development System documentation set; all man pages are available online through the **man**(C) and **scohelp**(XC) commands.

Character 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. Shipped as part of the Development System documentation set.

Schwaderer, W. D.

C Programmer's Guide to NetBIOS. Howard W. Sams and Company, 1988.

- Shirley, John. *Guide to Writing DCE Applications*. Sebastopol, CA: O'Reilly & Associates, 1992. A hands-on programming guide to OSF's Distributed Computing Environment (DCE) for first-time DCE programmers. Designed to help new DCE users make the transition from conventional, nondistributed applications programming to distributed DCE programming. Includes programming examples.
- 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. *Advanced Programming in the UNIX Environment*. Reading, MA 1992. A general purpose programming guide that presents information about using the system calls and C library routines that are common to most UNIX systems. The book provides information about the features that are specified in the ANSI C standard, the XPG standard, and the POSIX 1003.1 standard, with special emphasis on the USL UNIX System V, Release 4 and the 4.4 BSD release. The book includes a large number of code examples, including code for a database library, a program that communicates with a PostScript printer, a modem dialer, and pseudo terminals. Each chapter ends with a set of exercises to test and reinforce the knowledge learned in the chapter; Appendix C contains solutions to these exercises.
- 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.
- Taylor, Dave. *Global Software*. Discusses the issues in creating software for the growing international market including cultural and linguistic differences affecting the user interface to monetary, time, and date standards. The book provides information about planning and programming (with an extensive collection of actual program examples) as well as about marketing and presentation. Design errors and common pitfalls are pointed out to help readers avoid the same mistakes. The

author also discusses export restrictions and other aspects of the politics surrounding international distribution. A clear and understandable explanation of what types of encryption software can and cannot be legally distributed outside of the United States is an added bonus to this work.

Terplan, Kornel.

Effective Management of Local Area Networks. New York, NY: McGraw-Hill, 1992. Provides information about network management strategies, standardization, tools, and products for network designers, LAN operators, and people who select components and technology for LAN's, MAN's, and WAN's. Includes information about solving problems, automating database support, and maintaining uninterrupted operations.

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.

Periodicals

C++ Magazine. SIGS Publications, 588 Broadway, Suite 604, New York, NY, 10012.

Hotline on Object-Oriented Technology.

SIGS Publications, 588 Broadway, Suite 604, New York, NY, 10012.

International OOP Directory.

SIGS Publications, 588 Broadway, Suite 604, New York, NY, 10012.

Journal of Object-Oriented Programming.

SIGS Publications, 588 Broadway, Suite 604, New York, NY, 10012.

Object Magazine.

SIGS Publications, 588 Broadway, Suite 604, New York, NY, 10012.

SCO Magazine: The Journal for Builders

& Buyers of SCO Open Systems. CMP Publications, Inc., 600 Community Dr., Manhasset, NY 11030.

The Smalltalk Report.

SIGS Publications, 588 Broadway, Suite 604, New York, NY, 10012.

UNIX Review. 600 Harrison Street, San Francisco, CA 94107.

UNIX World.

The X Resource. O'Reilley and Associates, 103A Morris Street, Sebastapol CA 95472-9902. A quarterly working journal for X programmers that provides practical, timely information about the programming, administration, and use of the X Window System. This is the official publisher of the MIT X Consortium Technical Conference Proceedings, which form the January issue.

The X Journal: Serving the X Window System Community.

SIGS Publications, 588 Broadway, Suite 604, New York, NY,
10012.

SCO resources

The Santa Cruz Operation, Inc. offers various programs of special interest to developers. These are summarized here.

SCO Software Support.

SCO Developer Alliance.

Provides a special level of support for OEM's, VAR's, ISV's, and other resellers located in the United States and Canada. Membership is offered at two levels: Partners and Associates. Partners membership include 24-hour access to SOS (an online bulletin board and technical reference service), access to the SCO Developer Technical Resource Library, a subscription to *DiSCOver*, and discounts on various classes, seminars, and conferences. For more information, send email to uunet!sco!devrel or devrel@sco.COM, or call (800)726-8649.

SCO Forum

A week-long gathering for developers and marketing professionals working with SCO systems. Held on the campus of the University of California at Santa Cruz, this conference is a chance to confer with SCO staff and learn about current and future technologies.

Symbols, numbers

\$*, 248

486 architecture, bibliography, 310

A

accessory, Paint, 243

activate, alt_activate. *See* trigger

active window, 262

Aho, Alfred V., 307

animating icons, 252

API

color, 240, 261

defined, 239

display resources, 239, 257

icon object, 239, 242

session manager, 240, 266

app-defaults, 259, 260, 263, 266

application

character-based, 240, 247, 267

icon, 242

rules. *See* rules

starting, 247, 255

user-installable, 251

application programming interface. *See* API

apprules, 250

archiveMedia attribute, 283

archives

creating, 283

directory, 278

Arthur, Jay, 307

asm, 215

macro, 216

usage example, 216-217

usage guidelines, 220-221

assembly language escapes. *See* asm

ATTACH phase, 286, 288

attributes

archiveMedia, 283

determining SSO file attributes, 276

flags, 302

non-shared, 276

private, 276

public, 276

attributes (*continued*)

shared, 276

upgradeMapping, 298, 300

upgradeVersions, 298, 299

audit.h header file, 64

awk, bibliographic entry, 307, 309

B

Bach, Maurice J., 308

Banahan, Mike, 308

base address, 197

Bent, Wilson H. Jr., 308

Bentley, Jon Louis, 308

binary, compatibility, 241

bit-fields, 56

promotion of, 30

bitmap icon, 245

black-and-white bitmap, 245

Bolsky, Morris I., 308

Bourne, S. R., 308

Boyle, R.D., 311

Brodie, Jim, 312

Brown, Doug, 311

C

C++, 315

bibliographic entry, 309, 312, 314, 315

C programming language, bibliographic entry, 315

C2 security, 63

callback routine, 269

ccs (component control script), 285

environment variables, 296

example script, 294

relinking kernel, 292

syntax, 285

ccs(CDMT), 285, 291

CCS_PERSISTENT_STORAGE

directory, 289

variable, 291, 296, 297, 299

restrictions, 296

ccsSetup(CDMT), 291

CDMT (Custom Distribution Mastering Toolkit)

- advanced software installations, 285
- component description file, 272, 279
- component identifiers, 274
- creating custom-installable distributions, 272
- input files, 272, 278
- introduced, 271
- package description file, 272, 279
- phases, 286
- planning product structure, 275
- product description file, 272, 278
- sample make install script, 284
- setting up distribution tree, 277
- setting up environment, 278

cdmtArchive(CDMT), 278, 283

cdmtCompress(CDMT), 278, 282

cdmt.config file, 278

cdmt.config(CDMT), 272, 278, 283

cdmtConvert(CDMT), 279

CDMT_DIR variable, 278

cdmtInput(CDMT), 278, 279, 298

cdmtParse(CDMT), 278, 282

character-based applications, 240, 247, 267

Christian, Kaare, 308

CLASS, 259, 265

CLIENT

- directory, 276
- keyword, 304

cntl directory, 291

COFF, understanding and using, 309

COLOR, 259, 265

color

- API, 240, 261
- cells, 265
- control, 261, 263
- default, 258, 263
- defining, 262, 263, 264, 265
- display-dependent, 264
- for icons, 243, 245
- limitations, 265
- name variable, 262
- resources, 263
- selector, 263
- transparent, 243

command options, 246

compatibility, backward, 241

component description file, 279

- example, 280

component identifiers, 274

component scripts

- environment variables, 296
- guidelines, 291
- introduced, 285
- keywords, 297
- relinking kernel, 292
- structure, 293
- trapping interrupts, 292

componentCode, defined, 274

componentID, defined, 274

components

- dividing into packages, 275
- introduced, 273

CONFIGURE phase, 286, 290

const, 36-38, 55

constants, promotion of integral, 30

control, Color, 261

cpio(C), 283

cqs (component query script), 285

- environment variables, 296
- syntax, 285

cqs(CDMT), 285, 291

curses, explained, 309

curses(S), running under ksh job control, 126

custom(ADM), 250, 271, 284, 285

- creating installable distributions, 272
- upgrading software, 298

customquery(ADM), 292

customSched(CDMT), relinking kernel, 292

D

DAC (Discretionary Access Control), 66

data representation, 170-171

Date, C.J., 309

DCE, bibliography, 312, 313

deskconfobj tool, 250, 251, 257

deskshell, 247

desktop

- Color control, 261
- command language, 247
- icons. *See* icon
- Paint accessory, 243
- rules, 249

Developer Alliance Programs, 316

directories
 archives, 278
 CCS_PERSISTENT_STORAGE, 289
 CLIENT, 276
 cntl, 291
 /ibin, 288, 292
 input, 278, 279
 SHARED, 276
 sso, 278
 SSOroot, 273
 directory
 icon object, 250
 object, 242
 rules, 250
 display
 attribute symbols, 259
 color capability, 264
 grayscale, 263, 265
 monochrome, 263, 265
 resolution, 257, 258, 259
 resources. *See* resources
 resources API, 257
 display resources, API, 239
 distribution tree, SSOroot, 277
 distTreeRootCLIENT attribute, 277
 distTreeRootSHARED attribute, 277
 DOS window, 247, 248
 Dougherty, Dale, 309
 Dowd, Kevin, 309
 drop, alt_drop. *See* trigger
 drop trigger. *See* trigger
 dynamic linking, implementation, 192,
 203-213
 \$dynamic_args, 248

E

editor, icon, 243
 ELF, *See also* object files
 ellipsis notation, 23, 25, 55
 Ellis, M., 309
 encryption, 97
 Event manager API, 113
 executable files, 169
 EXPORT phase, 286, 290
 expressions, grouping and evaluation in,
 48-51

F

Feit, Sidnie, 309
 file
 data, 248
 icon, 248, 255
 installing, 250
 files
 cdmt.config, 278
 component description, 272, 279
 creating CDMT input files, 278
 determining SSO file attributes, 276
 /ibin/ccsSetup.sh, 291
 non-shared, 273, 276, 288
 package description, 272, 279
 private, 276, 301
 product description, 272, 278
 public, 276, 301
 shared, 273, 276
 /usr/include/sso/pathMap.h, 302
 /usr/lib/libssso.a, 302
 find(C), 279
 Fisher, Gerry, 312
 flags attribute, 302
 floating point operations, 9-19
 compares, 18
 data representation, 10-14
 denormalized numbers, 12
 double-extended, 17
 double-precision, 10
 exception handling, 15
 extended-precision, 11
 floating point to integer conversion, 18
 IEEE requirements, 18
 infinities, 13
 infinities I/O, 19
 maximum and minimum values, 12
 NaNs, 13
 NaNs I/O, 19
 normalized numbers, 11
 rounding, 14
 single-precision, 10, 16-17
 special-case values, 12
 square root, 18
 sticky bits, 15
 trap bits, 15
 unordered condition, 18
 font, default resources, 258
 Forum, SCO, 317
 Friedman, H. George Jr., 312
 function prototypes, 22-25

functions

- SSO_CLIENT, 303
- SSO_GetAbsolutePath, 303
- SSO_GetSegmentRootPath, 303, 304
- SSO_NO_ABORT, 303
- SSO_SHARED, 303

functions with varying argument lists, 25-27

G

- Gauthier, R., 309
- geometry, restoring, 267
- Gircys, Gintaras R., 309
- global symbols, 189-190
- Gofton, Peter W., 309
- Goodheart, Berny, 309
- grayscale display, 263, 265

H

- Harbison, Samuel P., 309
- HEIGHT, 259
- hold trigger. *See* trigger

I

- /ibin directory, 288, 292
- /ibin/ccsSetup.sh standard library, 291
- icon

- animated, 252
- API, 239, 242
- application, 242
- bitmap, 245
- border, 243, 245
- colors, 243, 245
- data file, 255
- directory, 250
- editor, 243
- existing, 243
- file, 248
- installing, 250, 257
- label, 243, 245, 250, 253, 257
- localized, 253
- menu, 248
- monochrome, 245, 246
- object, 242, 250
- picture, 243
- pixmap, 243

icon (*continued*)

- removing, 251
- rules, 241, 251, 253, 255
- scripts, 246, 247, 248
- size, 243, 245
- template, 243
- trigger, 246

icon label, 250

idinstall(ADM), 290

ifiles, 223

incomplete types, 51-53

init scripts, 250

input directory, 278, 279

input files, creating, 278

installation

- by user, 251
- interactive, 243
- of icons, 250, 257
- of object scripts, 250
- of pixmaps, 250
- of rules, 250
- script, 250, 260
- tool, 250

integral constants, promotion of, 30

Intel, bibliography, 310

Internationalization, general, 314

internationalization, 39-42, 45-48

interrupts, trapping, 292

J

Job control, ksh, 125

Journals, 315

K

Kenney, David, 312

kernel, relinking, 292

Kernighan, Brian W., 307, 310

keywords

- CLIENT, 304
- component scripts, 297
- OLD_CUSTOM_UPGRADE, 289, 297, 299
- SHARED, 304
- UPGRADE, 289, 298, 299

Koenig, Andrew, 311

Korn, David G., 308

Korn shell
 command and programming language, 308
 ksh, 125
 ksh, job control, 125
 ksh(C), 291

L

label, 250
 <label>.obj, 250
 LAN, bibliographic entry, 314
 \$LANG, 253
 Lapin, J.E., 311
 LD_BIND_NOW, 206
 Levine, John, 311
 Lewine, Donald, 311
 lex, bibliographic entry, 311, 312
 lex(CP), bibliographic, 311
 libraries
 libelf, 169
 shared, 241
 shared object, 192, 192-213
 static, 241
 link editing, 187-192, 203-213
 dynamic, 192, 203-213
 multiply defined symbols, 189-190
 linking
 to libraries, 241
 with SCO OpenServer Development System, 263
 LOAD phase, 286, 287
 local symbols, 189-190
 locale, 45, 47
 changed functions, 47
 new functions, 47
 Localization, general, 314
 localization, 251
 localized icon labels, 253
 localized rule file, 251
 Lomuto, Ann Nicols, 311
 Lomuto, Nico, 311
 long, 23

M

macro, asm, 216
 macro expansion, 34
 macros, SSO_ALLOC_ID_OBJECT, 302
 Magazines, 315

mapfiles
 defaults, 231
 error messages, 235
 example, 229
 map structure, 233
 mapping directives, 227
 segment declarations, 225
 size-symbol declarations, 229
 structure, 224
 syntax, 224
 usage, 224
 Mason, Tony, 311
 memory allocation, 241
 menu, icon pop-up, 248
 menu. *See* trigger
 Miller, Chris D.F., 311
 MIT Consortium Technical Conference, 316
 monitor. *See* display
 monochrome display, 263, 265
 Motif
 compliance, 241, 247
 libraries, 241
 toolkit, 240, 241
 Mouse, controlling in an application, 113
 mouse button
 See also trigger
 names, 246
 multibyte characters, 39-42
 multiply defined symbols, 189-190

N

NetBIOS, 313

O

object
 directory, 242, 250
 icon, 242
 scripts, 246, 247, 248
 object Builder tool, 243
 object files, 169-213
 base address, 197
 data representation, 170-171
 ELF header, 171-174, 177
 hash table, 212
 libelf, 169
 note section, 199-200
 program header, 194-200
 program interpreter, 204-206

object files (*continued*)

- program linking, 171-192
- program loading, 201-213
- relocation, 192
- section alignment, 180
- section attributes, 182-186
- section header, 177-186
- section names, 186
- section types, 180-186
- segment contents, 198-199
- segment permissions, 197-198
- segment types, 195-197
- string table, 187
- symbol table, 187-192
- tools for manipulating, 169

Object-oriented programming, 315

Open Desktop

- Release 1.1, 241
- Release 2.0, 241

ODR file, 257, 259, 260, 264, 266

OLD_CUSTOM_UPGRADE keyword, 289, 297, 299

OOP, bibliographic entry, 312

/opt area, 287, 291, 304

options, command, 246

O'Reilly, Tim, 311

OSF/Motif. *See* Motif

P

package description file

- example, 281
- introduced, 279

packageCode, defined, 274

packageID, defined, 274

packageName, defined, 274

packages, introduced, 273

paging, 201-203

paint accessory, 243

palettes, 261

Path Mapping API, 277

- C programs, 302

- introduced, 301

- shell scripts, 304

- Tcl scripts, 304

- using, 302

PATH variable, 288, 292

Performance, bibliography, 309

performance, 241

Periodicals, 315

phases, 286

- ATTACH, 286, 288

- CONFIGURE, 286, 290

- EXPORT, 286, 290

- LOAD, 286, 287

- REGISTER, 286, 289

- UNATTACH, 289

- UNCONFIGURE, 290

- UNEXPORT, 290

- UNLOAD, 288

- UNREGISTER, 289

picture, icon, 243

picture.px, 245

Pike, Rob, 310

Pinson, Lewis, 315

pixmap, icon, 243

PLANES, 259, 265

Plauger, P.J., 310

Plum, Thomas, 312

position-independent code, 203-210

preprocessing, 32-35

- stringizing, 34

- token pasting, 35

preprocessor directives, 257

product description file

- example, 280

- introduced, 278

products, dividing into components, 275

promotion, 28-31

- bit-fields, 30

- default arguments, 23

- integral constants, 30

- unsigned preserving, 28

- value preserving, 28

Protected subsystem, definition, 64

prot.h header file, 64

R

Rao, Bindu R., 312

regexp(TCL), 299, 300

REGISTER phase, 286, 289

relocatable files, *See also* object files

relocation, 192

removing

- icons, 251

- rules, 251

- reserved names, 42-45
 - for expansion, 44
 - for implementation use, 43
 - guidelines for choosing, 45
- resolution, display, 257, 258
- resources
 - API, 257
 - color defaults, 258, 263
 - color-dependent, 264
 - database utility, 258
 - default, 258, 260, 263, 266
 - font defaults, 258
 - resolution-dependent, 257
- Ritchie, Dennis M., 310
- Rochkind, Marc, 312
- Rosenberry, Ward, 312
- rules
 - data file example, 255
 - Desktop, 249
 - directory, 250
 - installing, 250, 251
 - label example, 254
 - localized, 251
 - menu example, 249
 - removing, 251
- Rutter, Andy, 308

S

- saveYourselfCB, 268, 269
- Schreiner, Axel T., 312
- Schwaderer, W.D., 313
- SCO Developer Alliance Programs, 316
- SCO Forum, 317
- SCO Software Support, 316
- scoActiveBackground, 262
- scoActiveForeground, 262
- scoActiveTopShadow, 262
- scoAltBackground, 262
- scoBackground, 262
- scoColor(XC), 263
- scoForeground, 262
- scoHighlight, 262
- scoterm, 247, 248, 267
- scoTopShadow, 262
- screen. *See* display
- script
 - init, 250
 - installation, 250
 - object, 246, 247, 248
- SecureWare, 64
- security.h header file, 64
- server, color support, 265
- session manager, 240, 266, 268
- setlocale(S), 46, 47
- SHARED
 - directory, 276
 - keyword, 304
- shared objects, implementation, 192, 203-213
- sh(C), 291
- shell scripts, Path Mapping API, 304
- Shirley, John, 313
- short, 23
- shutdown, 266, 269
- Signal handling, code example, 126
- signal handling, security implications, 64
- Smalltalk, 316
- SMT (Software Mastering Toolkit), 279, 299
- Sobell, Mark G., 313
- software management phases, introduced, 286
- Software Support, 316
- software upgrades, 298
- sso directory, 278
- SSO (Software Storage Object), 271
 - advantages, 274-275
 - creating archives, 283
 - determining file attributes, 276
 - introduced, 273
 - /opt area, 287
 - Path Mapping API, 277
 - introduced, 301
 - using, 302
 - /var/opt area, 288
- SSO tree, creating, 282
- SSO_ALLOC_ID_OBJECT macro, 302
- SSO_CLIENT function, 303
- SSO_CLIENT_ROOT variable, 291, 296
- SSO_EXPORT_TARGET variable, 291, 296
- SSO_GetAbsolutePath function, 303
- SSO_GetSegmentRootPath function, 303, 304
- SSO_NO_ABORT function, 303
- ssoPatch(CDMT), 283, 301
- ssoPathMap(CDMT), 302, 304
- SSOroot
 - distribution tree, 277
 - introduced, 273
- SSO_SHARED function, 303
- SSO_SHARED_ROOT variable, 291, 292, 296

SSO_UPGRADE_COMPONENTS variable, 297, 298
 startup, 268
 static trigger. *See* trigger
 Steele, Guy L., 309
 Stevens, W. Richard, 313
 STREAMS Primer, 307
 STREAMS Programmer's Guide, 307
 Stroustrup, Bjarne, 309, 314
 symbolic links, performance, 274

T

Taylor, Dave, 314
 TCB (Trusted Computing Base), 66
 Tcl scripts, Path Mapping API, 304
 TCP/IP, bibliographic entry, 309
 Terplan, Kornel, 314
 tokens, 32-35
 Tondo, Clovis L., 315
 toolkit
 Motif, 240, 241
 X Window System, 240, 241
 transition topics, 21
 transparent color, 243
 trigger
 activate, 246, 247
 d3, 246, 248
 defining, 246
 drop, 246, 248
 hold, 246, 248
 menu, 248
 static, 246, 247
 trigraph sequences, 32
 Trusted facilities, 63
 code examples, 67, 72, 90
 type qualifiers, 36-39
 types
 compatible and composite, 54-56
 incomplete, 51-53

U

UNIX window, 247, 248
 UNATTACH phase, 289
 UNCONFIGURE phase, 290
 UNEXPORT phase, 290
 UNLOAD phase, 288
 UNREGISTER phase, 289
 unsigned short, 23

UPGRADE keyword, 289, 298, 299
 upgrade mode, introduced, 298
 upgradeMapping attribute, 298, 300
 upgrades
 creating, 298
 introduced, 298
 specifying packages, 300
 specifying versions, 299
 upgradeVersions attribute, 298, 299
 /usr/include/sso/pathMap.h file, 302
 /usr/lib/libssso.a file, 302
 utilities. *See* accessory, control

V

varargs(S), 23
 variables
 CCS_PERSISTENT_STORAGE, 291, 296, 297, 299
 CDMT_DIR, 278
 component scripts, 296
 PATH, 288, 292
 SSO_CLIENT_ROOT, 291, 296
 SSO_EXPORT_TARGET, 291, 296
 SSO_SHARED_ROOT, 291, 292, 296
 SSO_UPGRADE_COMPONENTS, 297, 298
 /var/opt area, 288, 289, 291, 304
 vendorCode, defined, 274
 versionCode, defined, 274
 virtual addressing, 201-206
 volatile, 36-37, 38-39, 55

W

weak symbols, 189-190
 Weinberger, Peter J., 307
 Weiner, Richard S., 315
 wide character constants, 41-42
 wide characters, 40-42
 wide string literals, 41-42
 WIDTH, 257, 259
 window, active, 262
 WM_CLIENT_MACHINE, 268
 WM_COMMAND, 266, 269
 WM_PROTOCOLS, 268
 WM_SAVE_YOURSELF, 266, 268, 269

X

X Window System

Periodicals, 316

required, 241

toolkit, 240

utilities, 241

xdt3(XC), 253

.xdttdirinfo, 251

xrdb, 258

XSetCommand(XS), 267

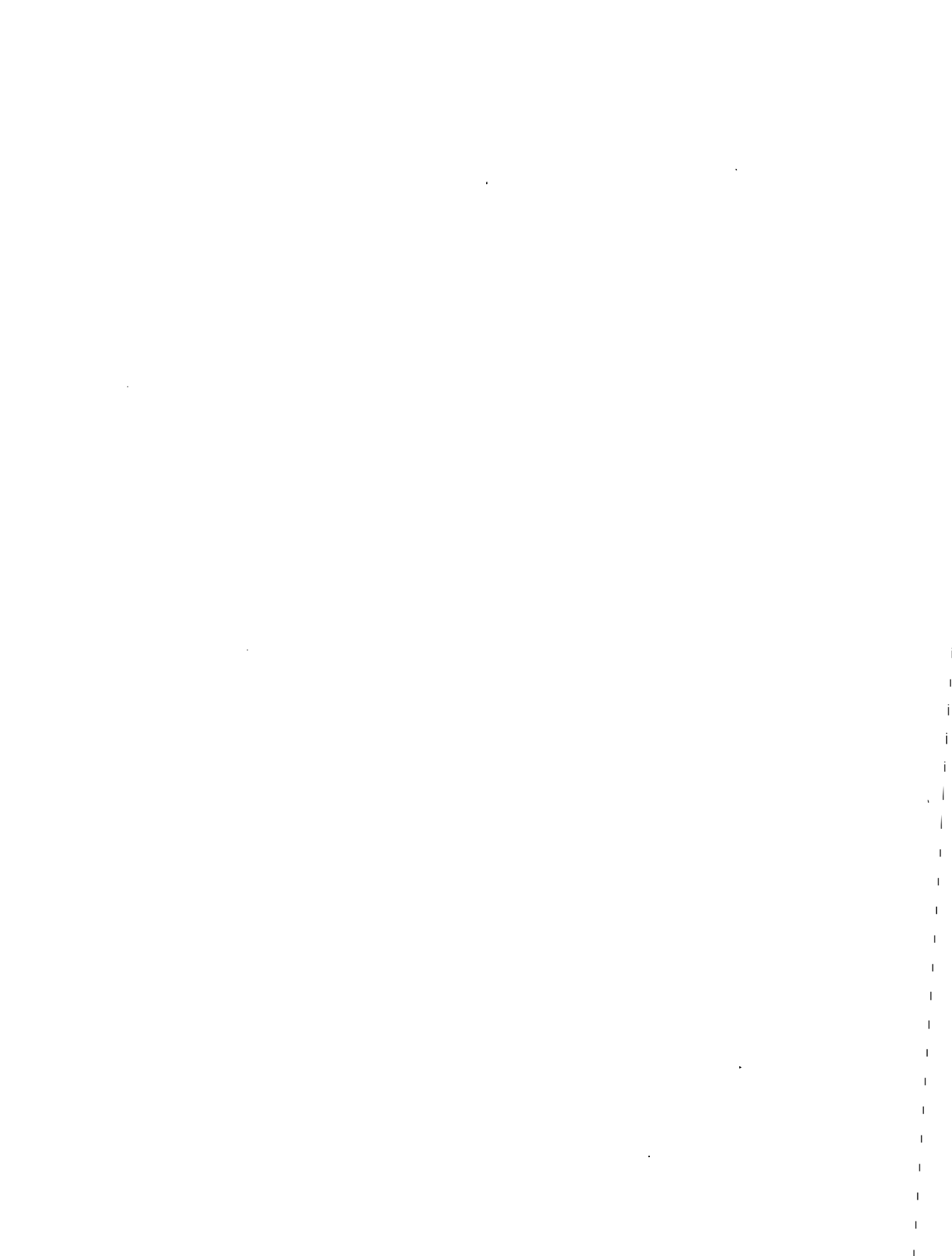
xt intrinsics, 241, 268

Y

yacc, bibliographic entry, 311, 312

yacc(CP), bibliography, 311

Young, Douglas A., 315



31 January 1995

AU10504P000



AU10504P000