

• • •
• •
•

SCO® UNIX® Development System

Developer's Topics



SCO[®] UNIX[®]

Development System

Developer's Topics



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

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

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

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

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

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

Date: 10 December 1991
Document Version: 3.2.4D

Preface **1**

Documentation conventions	2
Development System documentation set	3
Using the Documentation	4
Commercial books and articles	5

Chapter 1 ***Job control under ksh*** **7**

Chapter 2 ***Using trusted facilities*** **13**

Using protected subsystems	14
Guidelines for using trusted facilities	14
Programming examples	15
subsys.c example	17
prwarn.c example	22
loge.c example	40

Chapter 3 ***ISAM files*** **63**

ISAM concepts	64
Fields and records	64
Current position	64
Record number	65
Indexes	65
Primary and secondary indexes	65
Using indexes to access data	66
Multipart indexes	66
Unique indexes	66
Index compression	66
Index structures	67
Locking ISAM files and records	68

ISAM usage notes	70
Error handling	70
ISAM limits	71
ISAM applications on UNIX System V/386 runtime systems	71
Data types	71
Converting between integer data types	73
Converting between floating point data types	73
Creating an ISAM file	74
Adding data	76
Accessing data randomly	77
Updating records	78
Deleting records	79
Adding an index	79

Chapter 4

Constructing and using message catalogs **81**

Creating a message catalog	82
Code example using a message catalog	83
Combining the message catalog and the application program	85
Running the application program	86

Chapter 5

OMF/x.out format **87**

OMF Records	88
Record order	89
Numeric list of record types	90
Introduction to the record formats	90
Type representations for communal variables	117
The segmented x.out format	119
General description of x.out	120
Example of file layout	121
Iterated segments	122
Non-iterated segments and implicit bss	123
Large model	123
Special header fields	123

Symbol table	124
UNIX System V executable format	124
Selected portions of include files	125
Definition of terms	130

Chapter 6

C language compatibility with assembly language 141

C calling sequence for 8086/80286	141
Entering an 8086/80286 assembly routine	142
8086/80286 return values	143
Exiting an 8086/80286 routine	143
8086/80286 program example	144
80386 C-Language calling sequence	145
Entering an 80386 assembly-language routine	146
80386 return values	146
Exiting an 80386 routine	149
80386 program example	149

Chapter 7

Writing portable programs 151

Program portability	152
Machine hardware	152
Byte length	152
Word length	153
Storage alignment	153
Byte order in a word	154
Bit fields	155
Pointers	156
Address space	158
Character set	158
Compiler differences	159
Signed/unsigned char and sign extension	159
Shift operations	159
Identifier length	160
Register variables	160

Type conversion	160
Functions with a variable number of arguments	162
Side effects and evaluation order	162
Environment differences	163
Portability of data	164
Type-size summary	164
Byte-ordering summary	166

Chapter 8

Transport Layer Interface 167

Overview of the transport layer interface	167
Modes of service	168
Connection-mode service	169
Connectionless-mode service	173
State transitions	174
Connection-mode service	174
Local management	175
The client	177
The server	179
Connection establishment	182
The client	183
Event handling	184
The server	185
Data transfer	189
The client	191
The server	192
Connection release	194
The server	195
The client	196
Connectionless-mode service	197
Local management	197
Data transfer	199
Datagram errors	202
A read/write interface	203
write	204
read	205
close	205
Advanced topics	206
Asynchronous execution mode	206

Advanced programming example	207
TLI programming example summary	214
Connection-mode client	214
Connection-mode server	216
Connectionless-mode transaction server	220
Read/write client	222
Event-driven server	224

Preface

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

- Chapter 1 "Job control under ksh"
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 2 "Using trusted facilities"
provides three annotated examples of applications coded to run under the trusted facility software.
- Chapter 3 "ISAM Files"
provides tutorial information and examples of using the ISAM facility.
- Chapter 4 "Constructing and Using Message Catalogs"
discusses how to internationalize software using message catalogs. Included is an annotated example of constructing and using a simple message table within code.
- Chapter 5 "OMF/x.out format" discusses the internal details of the OMF/x.out format that is used for executable files (binaries) that are created with the -xenix option to the cc(CP) command.
- Chapter 6 "C language compatibility with Assembly language" gives information required to call assembly language routines from C programs, and C routines from assembly language programs.

Chapter 7 “Writing portable programs” discusses hardware topics that might impact source code portability.

Chapter 8 “Transport Layer Interface” provides detailed information about the UNIX System Transport Layer Interface, along with associated examples. This chapter discusses both connection-mode and connectionless-mode service for client and server.

Documentation conventions

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

Font conventions

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

Development System documentation set

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

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

The books included with the Development System are described here.

Developer's Overview

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

Programmer's Reference Manual

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

Encyclopedia

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

Developer's Topics

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

Programming Tools Guide

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

Debugging Tools Guide

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

User Interfaces Guide

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

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

Device Driver Writer's Guide

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

Macro Assembler Writer's Guide

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

Using the Documentation

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

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

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

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

Commercial books and articles

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

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

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

Marc Rochkind, *Advanced UNIX Programming*.

W. Richard Stevens, *UNIX Network Programming*.

Maurice Bach, *Design of the UNIX Operating System*

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

Chapter 1

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 the “Job control” article in the *Encyclopedia* and David Korn’s *Korn Shell* book. To understand this sample code, you should also understand the signal mechanism, which is discussed in the “Signals” article in the *Encyclopedia*.

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( SIGTTIN, &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             printw( "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 2

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 a specific criterion in its security policy, accountability, assurance, testing, and documentation.

All UNIX operating systems provide some measure of security by requiring 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 the first program in this chapter, **subsys.c**, 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. Remember to check the availability of system resources. For example, verify that enough processes are available and that you are not out of filesystem space.
- 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.

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(ADM)</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 |

log.e.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). 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 "Orange Book" and "Green Book," and SCO's *System Administrator's Guide* and the Development System *Encyclopedia*.

subsys.c example

The **subsys.c** program is similar to **auths(ADM)**, 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(F)** structure, taking into account 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 -lcrypt_i -o subsys
```

Example 2-1 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 2-2 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.
- 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 the `authaudit(S)` manual page 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 subsys
chmod 2111 subsys
```

Compile this program with the following command:

```
cc prwarn.c -lprot -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 2-3 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 <sys/security.h>   /* various security definitions */
7 #include <sys/audit.h>     /* audit definitions      */
8 #include <prot.h>          /* database definitions    */
9 #include <pwd.h>           /* /etc/passwd definitions */
10 #include <time.h>          /* localtime(S) definitions */
11 #include <utime.h>         /* utime(S) definitions    */
12 #include <sys/stat.h>      /* stat(S) definitions     */

13 extern int      getopt(int, char **, char *), optind, defopen(char *);
14 extern char     *optarg, *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 6-8 These lines include the three header files that define the trusted facilities.
- line 9 *pwd.h* defines the **passwd** structure. Trusted software uses this information to identify the user.
- lines 10-11 These header files define time conventions used to calculate the time between warnings and the time until the password expires.
- lines 13-14 The **getopt(S)** function checks and breaks up options in command lines for parsing by a C program. **defopen** and **defread** are used to read default program options. See **defopen(S)** for more information.

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

```

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

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

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

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

```

- lines 15-19 These global **define** 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 **#include** statements in all programs for the application.
- lines 15-16 The literals **TRUE** and **FALSE** are used to improve readability.
- line 17 **ARBSTRSIZ** is used in lines 82 and 297.
- lines 18-19 The time resolutions for this program are in seconds. Defining constants for these time spans increases the readability of the code.
- line 20 This line provides storage for the current date and time. The **time_t** type is defined in the `/usr/include/sys/types.h` file.
- line 21 This line is **TRUE** if **LUID** (logged in user identification) is accounts administrator.
- line 22 This line specifies the name of the file used to track the previous warning time for this user.
- line 23 This line specifies the name of the file used to store system defaults for the program.
- line 24 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 18. Lines 46-48 override this default value.
- line 25 This line defines a global flag.
- lines 26-49 These lines parse the `<hhmm>` string using standard string comparison functions.
- lines 44-45 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 could results only if the user made a typing error when issuing the command. When this routine is called (line 316), an error return causes it to call the **usage** routine defined in lines 271-290. **usage** prints an error message, a reminder of the correct syntax of the command, and then exits.

Example 2-5 prwarn.c (Part 3/13)

```

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

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

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

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

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

```

lines 52-66 These lines parse the *<days>* string.

lines 67-74 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. For more information, refer to the “Dates” and “Time” articles in the Development System *Encyclopedia*.

Example 2-6 prwarn.c (Part 4/13)

```

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

81     static char buf[ARBSTRSIZ];

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

```

lines 75-109 These lines calculate the relation between the specified date and today.

Example 2-7 prwarn.c (Part 5/13)

```

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

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

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

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

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

```

- | | |
|---------------|---|
| lines 110-121 | 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 <i>\$HOME/prwarn_time</i> file is used as the last time the user was notified. |
| lines 116-117 | The home directory is treated relative to <i>/</i> , even if the listing in the <i>/etc/passwd</i> file is not absolute. |
| lines 118-119 | This code uses the <i>stat(S)</i> system call to return statistics on the file; the <i>st_mtime</i> value returned is the last modification time stored in the file's inode. |
| lines 122-132 | 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 <i>last_reported</i> routine in lines 110-121. |

Example 2-8 prwarn.c (Part 6/13)

```

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

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

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

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

```

lines 133-146 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 147-160 This code calculates the time that must elapse between password changes.

Example 2-9 prwarn.c (Part 7/13)

```

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

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

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

181     return (0);
182 }

```

- lines 161-174** This code calculates the date or time after which the password cannot be used.
- lines 175-182** This code calculates the date or time at which the password was last changed.

Example 2-10 prwarn.c (Part 8/13)

```

183 enum password_change {
184     CanBeChanged,
185     TooEarly,
186     Procrastinated
187 } changable(
188     register struct pr_passwd *pr_pw
189 ) {
190     time_t lastchg, life;

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

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

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

197     return (CanBeChanged);
198 }

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

```



lines 183-198	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 197 and 258; TooEarly is referenced in line 194; Procrastinated is referenced in lines 196 and 261.
lines 191-192	If the password was never set, it is legal to set it.
lines 193-194	If the time since the last change was less than the "minimum time change" calculated in lines 147-160, the password cannot be changed.
lines 195-196	If the password has a finite lifetime and was last changed earlier than the date calculated in lines 161-168, the password cannot be changed (and the user cannot log in).
line 197	Otherwise, the password can be changed. This code makes no provision for the possibility that the user has retired or left the company.
lines 199-212	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 230.
lines 203-204	If the LUID has the auth subsystem authorization, the password may be changed.
lines 205-206	If the user has a pswduser , the LUID can change the password if it is the user's pswduser .
lines 207-208	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 209-210	Users can change their own passwords.
line 211	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 2-11 prwarn.c (Part 9/13)

```

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

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

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

```

- | | |
|---------------|---|
| lines 213-270 | These lines report to the user if the password expires soon. |
| lines 220-229 | The user has an entry in the <i>/etc/passwd</i> file, so without the Protected Password entry, you can audit the problem. |
| lines 230-236 | If the user is not authorized to change the password, print an error message. The <code>authorized_to_change</code> subroutine is defined in lines 199-212. |

Example 2-12 prwarn.c (Part 10/13)

```

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

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

```

- lines 237-238 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 239-270 Otherwise, issue a warning that the password must be reset soon, and update the time of the last warning.
- line 254 Lines 76-109 define the **when** subroutine. The subroutine returns a human-readable string representing a number of day in the future or past.

line 258	Line 184 defines the CanBeChanged case.
line 261	Line 186 defines the Procrastinated case.
line 267	Lines 122-132 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 2-13 prwarn.c (Part 11/13)

```

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

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

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

282         putc('0', stderr);
283     }
284     (void) fprintf(stderr,
285         "Usage: %s [ -d days ] [ -t hh:mm ] [ users ]...\n",
286         command_name
287     );
288     exit(2);
289     /* NOTREACHED */
290 }

```

lines 271-290 This subroutine prints the usage message and exits the program if the user issues the command incorrectly. It is called in lines 312, 316, and 320.

Example 2-14 prwarn.c (Part 12/13)

```

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

298     set_auth_parameters(argc, argv);

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

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

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

```

- line 291 This line begins the **main()** routine. Thus far, everything has been subroutines that are called as functions in **main()**.
- line 298 **set_auth_parameters(S)** must be the first function called in **main()**.
- lines 299-300 These lines initialize the current date and time, and determine whether or not the logged-in user is an accounts administrator.

- lines 301-307 If there are any legal system-wide defaults, use them rather than the program's built-in defaults (lines 24-25 and 50-51).
- lines 308-322 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 323-348 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 323-335 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 336-348 If the named user is not listed in the */etc/passwd* file, then there is no user by that name. In this case, print an error message to the user's terminal.

Example 2-15 prwarn.c (Part 13/13)

```

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

```

line 323 This line checks **optind** from line 13. **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 324-325 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 213-270) to print a report about the current user. If it fails, it audits the fact and prints error messages.

loge.c example

The `loge.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. `loge.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 `loge.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(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

Example 2-16 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 <sys/stat.h>       /* stat(S) definitions     */
9 #include <time.h>           /* localtime() definitions  */
10 #include <sys/security.h>   /* various security definitions */
11 #include <sys/audit.h>      /* audit definitions        */
12 #include <prot.h>           /* database definitions      */

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

15 #define TRUE      1
16 #define FALSE     0

17 #define ARBSTRSIZ 300      /* ARBiTrary STRing SIze      */
18 #define MAXCMDLEN ARBSTRSIZ /* MAXimum CoMmanD name LENgth */

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

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

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

```

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

```

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

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

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

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

```

Example 2-18 loge.c (Part 3/17)

```

48     set_auth_parameters(argc, argv);

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

```

line 48 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 49-64 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 153-163 define the **failed** subroutine.

Example 2-19 loge.c (Part 4/17)

```

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

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

```

lines 65-72 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 73-78 These lines check that the specified command exists in the *valhalla* directory and that it is executable.

Example 2-20 log.c (Part 5/17)

```

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

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

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

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

```

lines 79-86 These lines check that the file conforms to the specification listed in the File Control database.

lines 87-100 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 101-116 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 111 Empty cipher text is the same as no password.

Example 2-21 log.e.c (Part 6/17)

```

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

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

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

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

```

lines 117-139 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.

-
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, 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.

- 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.
- line 117 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 45 and 46.
- lines 122-124 If the user supplies the correct password, a message is sent that the use can execute this command as super user.
- line 139 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 166-206 for the definition of the **su**log function.
- lines 140-152 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 143 The **audit_no_resource(S)** routine also prints a message to **stderr**.
- lines 144-145 **setgid(S)** is a system call that is audited automatically. If an error occurs, it is handled with the **syserr(S)** routine rather than an audit record.
- lines 146-147 Similarly, **execv(S)** is also audited automatically.

Example 2-22 loge.c (Part 7/17)

```

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

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

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

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

```

lines 166-206

These lines define the **sulog** function called in line 139. 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 reliable 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 179-189 These lines scan */etc/default/su* for definitions of *SULOG* 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 190-195 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 196-199 These lines log this attempt in the super-user logfile.
- lines 200-205 If *CONSOLE* is defined and the user is not at that location, log this attempt on that terminal.
- lines 197, 203 The *log_write* function is defined in lines 208-232.



Example 2-23 loge.c (Part 8/17)

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

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

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

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

Example 2-24 log.c (Part 9/17)

```

207 void    graphic(char *, FILE *);

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

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

```

lines 208-232 This `log_write` function is called in lines 197 and 203. 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 228 Lines 223-254 define the `graphic` function which prints a string using all readable, printable ASCII characters.

Example 2-25 loge.c (Part 10/17)

```

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

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

```

lines 233-254 Line 228 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 2-26 loge.c (Part 11/17)

```

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

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

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

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

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

```

lines 255-273 These lines open a log file.

lines 274-291 These lines convert a `create_file_securely( )` return value to a message.

Example 2-27 loge.c (Part 12/17)

```

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

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

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

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

```

lines 292-309 The **syserr** subroutine is used rather than the standard **perror(S)** routine because **syserr** allows the use of **printf** format strings, and **perror** does not.

Example 2-28 logex.c (Part 13/17)

```

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

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

```

- lines 310-326 The **check_TCB_path** subroutine and the **check_DAC_path** routine (defined in lines 327-383) 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 312-314 The subroutine takes three arguments, specifying, in turn, the full pathname of a file that is listed in the File Control Database (line 312), the file type (line 313; valid values are listed in lines 25-27), and a description of the file.
- line 312 The *pathname* is assumed to be both canonical, meaning it does not contain consecutive backslash (\) characters, and absolute, meaning that it begins with slash (/).
- line 325 **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 2-29 loge.c (Part 14/17)

```

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

```

lines 327-383 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 310-326, this routine verifies the integrity of the File Control database itself.

lines 336-339 These lines make a working copy of the pathname that can be modified.

lines 340-383 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 2-30 loge.c (Part 15/17)

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

382     return (0);
383 }
```

Example 2-31 loge.c (Part 16/17)

```

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

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

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

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

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

```

- lines 384-430 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 396-399 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 400-419 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 2-32 loge.c (Part 17/17)

```

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

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

429     return (WrongDAC);
430 )

```

lines 420-430 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 3

ISAM files

ISAM (Indexed Sequential Access Method) is a defined X/Open standard for a C interface that allows programmers to do search operations on files by record and/or field. SCO ISAM is a library of routines that enable a programmer to build custom data management procedures into any type of application. It gives the programmer precise control over how data is handled, and so can be used when the commercial database management systems do not provide the appropriate data handling for the application or when the data handling portion of the application is too small to warrant the resources required by a full-scale database management system.

The SCO ISAM package is based on the X/Open standard with the following enhancements:

- Case sensitivity may be turned off in string searches.
- Routines to generate statistics on the use of ISAM files are provided.
- Alternate, international collating sequences are available.
- Functions are provided to generate unique data values.
- Shared locking is supported.

For more detailed information about using SCO ISAM, see the appropriate manual pages.

This chapter discusses the ISAM file capabilities and provides examples of programs that access ISAM files. The "ISAM" article in the *Encyclopedia* lists the Section S manual pages that document the ISAM function calls, and explains how to run an application that uses ISAM files on a Runtime UNIX System V/386 release.

ISAM concepts

It is helpful to understand some basic concepts and terminology before designing an application that uses ISAM files.

Fields and records

Data files to be stored using ISAM must be structured into fields and records. A field is a category of data, such as a name or an address. A data file can have multiple fields and any number of records. A record is one set of data values for all the fields describing a specific subject. In a file that stores address information, for example, each person listed might have a record comprised of data values filling name, address, and phone number fields.

When data is put into table format, fields comprise the columns of the table and records comprise the rows. Unlike a database management program, ISAM does not include a facility for naming fields. Field names, if they are used, must be assigned and maintained by the user's program.

In an ISAM file, each field is assigned a fixed data length depending on the type of data it will hold. A record is stored as the set of the fields strung together. Thus, each record in an ISAM file also has a fixed length that is the sum of the lengths of all the fields in that file. The order and lengths of fields are established when the data file is created.

An ISAM file consists of a fixed header and data records. New records are usually appended to the end of the file, but may be inserted in the middle of the file if any vacant slots are left from previous deletions.

Current position

An open ISAM file's current position determines which record is retrieved next when `isread(S)` is used to read the file and select the next, current, or previous record. Note that this position is associated with the ISAM file descriptor (`isfd`), not the actual file itself. So if several file descriptors are simultaneously opened for one file, each will probably have a different current position.

The `isread(S)` and `isopen(S)` calls set the current position, as do `isstart(S)` (used to select an alternate index and to locate a record) and `iswrcurr(S)` (used to change the contents of a record). The current position is used by `isdelcurr(S)`, `isread(S)`, and `isrewcurr(S)`.

Record number

The record number is an identifier for the record last read, written, or selected with `isstart(S)`. This value is stored in the global variable `isrecnum`.

The value of `isrecnum` can be saved and used for direct access of the previously located record, to update it with `isrwcrr` or delete it with `isdelrec`. For example, you can read a record that you want to update into a buffer, save the `isrecnum` value in a variable, modify the data in the buffer, and then update the original record as identified by the record number. The actual value of `isrecnum` has little lasting significance.

`isrecnum` is updated by the following calls: `isdelcrr`, `isdelete`, `isdelrec`, `isread`, `isrewcrr`, `isrewrec`, `isrewrite`, `isstart`, `iswrcrr`, and `iswrite`.

Indexes

Indexes provide the primary access to the records in ISAM data files. An index can be thought of as a list of paired data values and locations, stored by data value. A calling program can use an index to locate a particular piece of data quickly.

In the simplest case, an index has one part (any single field from an ISAM file) and the data is indexed in ascending order. The data field that is indexed is called the index key. When you create an index, an entry is created for each record in the index key. An index entry consists of a record's key value and its location in the data file. ISAM function calls automatically update each index as records are added, updated, or deleted.

ISAM provides two methods for retrieving records: keyed sequential access and random access. Keyed sequential access (or just sequential access) retrieves records in order based on the current index. Random access retrieves records based on the value of a key; you specify a key value for the currently indexed file, and the file is searched for a record containing that value. Random access is useful when you need to locate specific information.

Primary and secondary indexes

Each ISAM file has at least one index, called the primary index or primary key, which is created when the data file is built. The primary index is most useful when it holds unique values so that each record can be identified without ambiguity.

An ISAM file may have more than one index, each of which is defined separately after the file is created. These additional indexes are called secondary indexes, and each provides a different route into the data file. An ISAM file can have any number of secondary indexes.

Using indexes to access data

An index is structured as a balanced tree (b+tree). A search for a specific record takes the shortest route through the branches of the tree to the desired key value. New index entries are positioned in the tree so that the index is always maintained in sorted order. This incremental reorganization ensures rapid access to your data.

Each ISAM index provides one sequence for accessing records, regardless of the physical order of the records. The name “indexed sequential access method” comes from this feature. The file appears to be in the order specified by the index. Because you can create several indexes on a data file, you potentially can locate records by any of the fields or combinations of fields. By opening the same file several times, you can use several indexes at once.

When the ISAM file is opened, the primary index is automatically used to locate records. With the `isstart(S)` call, you can select an alternate index to use when searching the file. The index being used to access data is called the current index.

Multipart indexes

An index can also have multiple parts, combining two or more fields from one ISAM file. Indexing on both last name and first name is an example of a multipart index. The index parts need not be contiguous; they can be at different locations in a record. Each of the parts can be set separately to use ascending or descending order when the index is defined. You can also specify whether case sensitivity should be used during searches of an indexed part with a character data type.

The maximum total length of any index is 120 bytes. A multipart index can include up to nine noncontiguous portions of a record.

Unique indexes

An index can be defined to accept or not accept duplicate values in the indexed field. When the index is defined to allow no duplicates, it is called a unique index, and each record can be uniquely identified by the data in the index key.

Index compression

You can also define indexes to use compression, meaning that the index data is condensed to save disk space. The trade-off is that compression can cause data access to be somewhat slower than with noncompressed indexes. Several types of compression are available.

Index structures

ISAM uses two C-program structures, **keydesc** and **keypart**, to specify an index. These structures are defined in the *isam.h* header file. Note that the sum of the lengths of all the parts cannot exceed **MAXKEYSIZE**, which is defined as 120 bytes in *isam.h*. The maximum number of parts in a key cannot exceed **NPARTS** (also defined in *isam.h*).

keydesc structure

The **keydesc** structure describes the index as a whole. It has the following members:

```
short k_flags;           /* duplicate and compression flags */
short k_nparts;          /* number of parts in this index */
struct keypart k_part[NPARTS]; /* array of key parts */
```

k_flags indicates whether this index allows duplicate values and/or uses compression of the index to save space. The different values of compression imply different levels of compression, which affect performance to different degrees. Values can be summed to combine options. Note that the last two values are sums of previous values and are provided for convenience. The values that can be used in **k_flags** are:

ISNODUPS	no duplicates allowed
ISDUPS	duplicates allowed
DCOMPRESS	compression of duplicates
LCOMPRESS	leading compression
TCOMPRESS	trailing compression
COMPRESS	all compressions
ISALL	all compressions and allows duplicates

No default value is defined, so **k_flags** must be initialized to some value. This means if no compression is desired, then either **ISNODUPS** or **ISDUPS** must be used.

k_nparts specifies the number of parts in the index. The number of parts cannot exceed **NPARTS**, an integer defined in the *isam.h* header file.

k_part an array of **keypart** structures, described below.

keypart structure

The **keypart** structure is nested in the **keydesc** structure and describes each part of an index. It has the following members:

```

short kp_start;          /* offset of first byte of this key part */
short kp_leng;           /* number of bytes in this key part */
short kp_type;           /* data type of this key part */

```

- kp_start** indicates the starting byte of a key part within a record. You must examine the data file definition to determine which byte this is for a particular field. Note that this is the offset into the record, so the first byte of a record is always zero.
- kp_leng** indicates the number of bytes of a key. For number fields, this is the length of the data type. For character fields, this can be all or part of the character field.
- kp_type** indicates the data type and sorting sequence of the field. By default, the index is in ascending order, and the character part of the key (if any) is case-sensitive during searches. The data types and sorting options are defined in the *isam.h* header file:

CHARTYPE	character data type
INTTYPE	short integer data type
LONGTYPE	long integer data type
DOUBLETTYPE	double-precision floating point
FLOATTYPE	single-precision floating point
ISDESC	descending sort order (used with any data type)
ISNOCASE	no case sensitivity (used only with CHARTYPE)

Locking ISAM files and records

On multiuser systems, data in ISAM files can be locked so other processes cannot alter it. This is useful when an operation is in progress and the user does not want the data changed until the operation is complete. One example is an airline reservation system, where a clerk might lock the record for an available seat while the customer makes a decision.

ISAM file and record locking is implemented using a shared memory region, access to which is controlled with System V semaphores. The kernel structures for the shared memory region and semaphores are created when the first user opens an ISAM file, and removed when the last user closes the last ISAM file. The *key* for these structures is based on the value of the **DBKEY**

environment variable; all users must have the same value for **DBKEY** to ensure that locking operations work correctly. Because ISAM uses shared memory and semaphores, locking operations may not be reliable when ISAM file access is done over NFS software.

On single-user operating systems, the lock functions have no effect. However, the code runs without error, so it is useful to include them so that the same application can run correctly on a multiuser system.

You can lock individual records in a file or the entire file. Individual records can be locked in one of the following modes:

- | | |
|-----------|---|
| automatic | A single record is automatically locked when it is read, and unlocked just after the next ISAM call is completed. Other processes cannot read, write, or lock the record until it is unlocked. |
| manual | Each record is explicitly locked and unlocked with function calls. Any number of records can be locked, and they may remain locked for as long as needed. Other processes cannot write or lock the record until it is unlocked. |
| shared | A single record is locked and unlocked explicitly. Other processes can read the record and acquire shared locks on it, but they cannot change or obtain exclusive locks on it until it is unlocked. |

You can also lock an entire file, with one of the following schemes:

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

The type of locking to be imposed is indicated by a mode flag used when the file is opened with the **isopen** or **isbuild** call.

ISAM usage notes

Note these guidelines when using ISAM files in your application:

- Include the *isam.h* file in any code that uses ISAM routines.
- Use `cc -lisam` when compiling programs that use the ISAM routines.
- Before you run a program that accesses ISAM files, you must set the `DBKEY` environmental variable to a value of "0". To set up `DBKEY` using the Bourne Shell, type the command `DBKEY=100`. To set up `DBKEY` using the C Shell, type the command `setenv DBKEY 100`.

Error handling

Code that uses ISAM routines must be written to handle error conditions that arise from the ISAM calls. Most ISAM calls return a value of zero to indicate successful execution and a value of -1 to indicate an error condition and unsuccessful execution.¹ When an error occurs, the global integer `iserrno` is set to indicate the nature of the error. The values of `iserrno` are listed in `intro(S)`. To make testing for errors easier, a set of macro names are defined to correspond to the values of `iserrno`. These constant names are listed in the *isam.h* header file and on the `intro(S)` manual page.

Two global status characters, `isstat1` and `isstat2`, provide diagnostics about the type of error. The `isstat1` variable indicates general information about the status of a function call. The `isstat2` variable provides further information, based on the value of `isstat1`. The value of these two variables also are listed in `intro(S)`.

Generally, you should test the return value from each ISAM function call for a -1 error condition. If true, then test for the likely values of `iserrno` or call a general-purpose error testing routine.

1. Some ISAM calls return a non-negative integer to indicate an error.

ISAM limits

The following limitations apply to all SCO ISAM files:

record length	no limit
number of records	no limit
number of ISAM files	10
number of open ISAM files	20
index length	120 bytes
non-contiguous index portions	9
number of indexes	no limit
number of locks	2000
number of values of a duplicate	no limit

ISAM applications on UNIX System V/386 runtime systems

Applications that use ISAM files will not execute on a UNIX System V/386 runtime system on which the SCO ISAM RTS is not installed. All current versions of SCO UNIX System V/386 include the ISAM RTS; it can be purchased separately for older releases of the operating system. The script that installs the ISAM application must include the `/usr/bin/inipcrm` file and give it the `root` superuser ID; `inipcrm` performs operations on shared memory segments for which users may not have privileges.

Systems on which ISAM applications run must be tuned to support the shared memory region and set of semaphores used by the ISAM routines. Applications should be coded to return meaningful error messages if these resources are not available.

Data types

Each field in an ISAM file is generally assigned one of the allowed data types. However, data types are significant only for index keys. Data of any type can be stored in the non-key fields of a record. It is good practice to keep a specific data type in each field of your ISAM file in case you want to add an index on that field later.

The allowed data types are:

ISAM data type	C data type	Length
CHARTYPE	char	variable
INTTYPE	short	2 bytes
LONGTYPE	long	4 bytes
FLOATTYPE	float	machine-dependent
DOUBLETTYPE	double	machine-dependent

Each ISAM data type is associated with a C-language data type. All number data types also have their lengths assigned in the header file. Note that the lengths of `FLOATTYPE` and `DOUBLETTYPE` are automatically matched to the corresponding machine data length. The length of character data fields is assigned by the programmer when a data file is created.

CHARTYPE The `CHARTYPE` data type is used to store character strings such as names and addresses. Numbers also may be stored in character format if no arithmetic operations are to be performed on them. Zip codes and telephone numbers are examples of such numbers.

Each field that uses `CHARTYPE` has a fixed length that is assigned when the data file is created. Null termination of character strings is not required.

INTTYPE and LONGTYPE

The integer data types `INTTYPE` and `LONGTYPE` are used to store 2-byte and 4-byte signed integers, respectively. If you are sure the data values for an integer field will be in the range -32768 to +32767, then you can use `INTTYPE`. Otherwise, use `LONGTYPE`.

FLOATTYPE and DOUBLETTYPE

The data types `FLOATTYPE` and `DOUBLETTYPE` are used to store single- and double-precision floating-point numbers, respectively.

These ISAM data types correspond to the C-language data types `float` and `double`. The length and format of these data types are machine dependent. The definitions of `FLOATTYPE` and `DOUBLETTYPE` in *isam.h* automatically adjust them to the sizes on the current machine.

A set of macros are defined in the *isam.h* header file to convert data between the ISAM format and the native machine format.

Converting between integer data types

The bytes that make up an integer are always stored in the ISAM file in the order you would expect: most-significant byte first and least-significant byte last. Some machines, however, store integers in the reverse order. Four routines are available to convert integers between ISAM and machine format:

- ldint(p)** returns a short integer in machine format. Here *p* is a **char** pointer to the first byte of the format **INTTYPE**.
- ldlong(p)** returns a long integer in machine format. Here *p* is a **char** pointer to the first byte of the format **LONGTYPE**.
- stint(i,p)** converts a machine-format short integer *i* to **INTTYPE** and stores it at location *p*. Here *p* is a **char** pointer to the first byte of **INTTYPE**.
- stlong(l,p)** converts a machine-format long integer *l* to **LONGTYPE** and stores it at location *p*. Here *p* is a **char** pointer to the first byte of **LONGTYPE**.

ldint and **ldlong** might be used when ISAM records are written to a temporary storage area for further processing by a C program. Any integers being passed from the buffer to the executing program are first converted to machine format with the **ldint** and **ldlong** routines. **stint** and **stlong** might be used when integers produced or processed by an executing C program are to be written to an ISAM file. The **stint** and **stlong** routines convert the machine-format integers to ISAM format and place them in a buffer in preparation for writing them to the ISAM file.

Converting between floating point data types

Integers in ISAM format need not align along word boundaries as must integers in some machine format. On certain machines, floating-point numbers must generally be aligned on word boundaries. ISAM-format numbers need not be. Four routines are available to convert between the unaligned ISAM format and the aligned machine format.

- ldfloat(p)** returns a **float** in machine format. Here *p* is a **char** pointer to the first byte of the format **FLOATTYPE**.
- lddbl(p)** returns a **double** in machine format. Here *p* is a **char** pointer to the first byte of the format **DOUBLETTYPE**.
- stfloat(f,p)** converts a machine-format **float** to **FLOATTYPE** and stores it at location *p*. Here *p* is a **char** pointer to the first byte of **FLOATTYPE**.
- stdbl(d,p)** converts a machine-format **double** *d* to **DOUBLETTYPE** and stores it at location *p*. Here *p* is a **char** pointer to the first byte of **DOUBLETTYPE**.

`ldfloat` and `lddbl` might be used when ISAM records are accessed for processing. Any floating-point numbers being passed from the buffer to the executing program must be first converted to machine format with the `ldfloat` and `lddbl` routines. `stfloat` and `stddbl` might be used when floating-point numbers produced or processed by an executing C program are to be written to an ISAM file. The `stfloat` and `stddbl` routines convert the machine-format numbers to ISAM format and place them in a buffer in preparation for writing them to the ISAM file.

Creating an ISAM file

The rest of this chapter consists of code examples that show how to create an ISAM file, add data to it, access data randomly, update and delete records, and add a secondary index. These examples create and modify an ISAM file for a book ordering application. Each record in this file contains the following fields:

Field	Data Type	Bytes
ISBN	long	4
Title	char	20
Publisher	char	20
Quantity	long	4
Price	long	4
Total length		52

The *ISBN* field is the International Standard Book Number, which uniquely identifies each book. This is used as the primary index.

To create an ISAM file with SCO ISAM, you fill an index structure with values and then use the `isbuild(S)` function call.

Here is a program fragment that creates an ISAM file with these specifications:

```

1 #define RECSIZ 52

2 struct keydesc key ;
3 int fdbook ;

4 key.k_flags = ISNODUPS ;
5 key.k_nparts = 1 ;
6 key.kpart[0].kp_start = 0 ;
7 key.kpart[0].kp_leng = LONGSIZE ;
8 key.kpart[0].kp_type = LONGTYPE ;

9 if ((fdbook = isbuild ("books", RECSIZ, &key, ISINOUT + ISEXCLLOCK)) << 0)
10 isclose(fdbook);

```

- | | |
|---------|---|
| line 1 | The RECSIZ constant is used in line 9 for the total record length of the ISAM file. |
| line 2 | The index key structure keydesc is defined in the <i>isam.h</i> header file. That file also holds the definitions of the macros ISNODUPS , LONGSIZE , LONGTYPE , ISINOUT , and ISEXCLLOCK . |
| line 3 | This is the file descriptor. |
| line 4 | Set the flag for no duplicate values. |
| line 5 | Specify number of key parts. Using a value greater than one makes this a multipart index that keys on two separate fields (such as the first name and last name field). |
| line 6 | position of first byte |
| line 7 | length of key part. LONGTYPE is an ISAM data type that is 4 bytes long. See the "Data types" section earlier in this chapter for a list of all valid ISAM data types. |
| line 8 | data type of keypart. LONGTYPE is the flag used to indicate long integer data type. See the "keypart section" earlier in this chapter for a list of all flags that can be used for this member. Note that no sort order is specified, so the file will be sorted in ascending order. |
| line 9 | Create the ISAM file named <i>books</i> . The total record length is RECSIZE . The ISINOUT + ISEXCLLOCK parameter opens the file for input and output in an exclusive lock mode. This is the most common value, but others may be used. |
| line 10 | Close the ISAM file. |

Adding data

To add data to an ISAM file:

1. Open the file with the `isopen` function call.
2. Fill a buffer with the data values.
3. Call the `iswrite` function.
4. Check for errors and close the file when finished.

This program fragment illustrates the steps for adding data to an ISAM file.

```

1 int fdbook ;
2 char buffer[RECSIZ] ;
3 long isbn ;
4 char *title ;
5 char *publisher ;
6 long quantity ;
7 long price ;

8 fdbook = isopen("books", ISINOUT + ISEXCLLOCK);
9 isbn = 0810462613 ;
10 title = "Programming in C " ;
11 publisher = "Hayden Books " ;
12 quantity = 2 ;
13 price = 2895 ;
14 stlong(isbn, buffer) ;
15 strcpy(buffer + 4, title) ;
16 strcpy(buffer + 24, publisher) ;
17 stlong(quantity, buffer + 44) ;
18 stlong(price, buffer + 48) ;
19 iswrite(fdbook, buffer) ;
20 isclose(fdbook) ;

```

line 1	fdbook is the file descriptor
line 2	buffer for new record
lines 3-7	data fields that will be populated
lines 14-18	Fill the buffer with values. The <code>stlong</code> function converts a long integer to ISAM format. It is defined in the <i>isam.h</i> header file.
line 19	Write the data to the ISAM file. In place of the <code>iswrite</code> function, you could use <code>iswrcurr</code> which writes the record and makes the new record the current record. The <code>iswrite</code> function does not make the new record the current record. Both functions also update all indexes defined in the file.
line 20	Close the ISAM file.

Accessing data randomly

The **isread** function reads a record from an ISAM file into a buffer. It can be used for sequential reads or random reads. To perform a random read:

1. Open the ISAM file.
2. Fill the key field with a search value.
3. Call the **isread** function with the **ISEQUAL** mode.

The following program fragment illustrates these steps, where the search value is stored in the location in a buffer corresponding to the key field. The record is then read into the same buffer:

```

1 int fdbook ;
2 char buffer[RECSIZ] ;
3 long isbn = 0810462613 ;

4 fdbook = isopen("books", ISINOUT + ISEXCLLOCK) ;
5 stlong(isbn, buffer) ;
6 isread(fdbook, buffer, ISEQUAL) ;
7 isclose(fdbook) ;

```

line 1	File descriptor
line 2	Buffer to hold the record
line 3	Search value
line 4	Open the ISAM file.
line 5	Insert search value
line 6	Read record into buffer
line 7	Close the ISAM file.

Once you have the record in the buffer, you can process the values as needed.

Updating records

In ISAM, updating means rewriting whole records. The steps are:

1. Open the ISAM file.
2. Use **isread** to locate the desired record.
3. Change the data values in the buffer.
4. Call **isrewcurr** to rewrite the current record.

This program fragment changes the book quantity from the previous example.

```

1 int fdbook ;
2 char buffer[RECSIZ] ;
3 long isbn ;
4 long quantity ;
5 fdbook = isopen("books", ISINOUT + ISEXCLLOCK) ;
6 isbn = 0810462613 ;
7 quantity = 10 ;
8 isread(fdbook, buffer, ISEQUAL) ;
9 stlong(quantity, buffer + 44 ) ;
10 isrewcurr(fdbook, buffer) ;
11 isclose(fdbook) ;

```

line 1	File descriptor
line 2	Buffer to hold selected record
line 3	Index key
line 4	Field to be changed
line 5-7	Open file and move to record that is to be updated.
lines 8-9	Read record into buffer and insert new value into buffer. If the index key used to locate the record is the primary key and does not allow duplicate values (as with this one), a record can be updated directly without reading it first. This is done by filling the buffer with the search value and new data values, and then calling isrewrite instead of isrewcurr .
line 10	Rewrite the current record. The isrewcurr function is more often used when the search key allows duplicate values. Because you cannot be sure which record a key will locate, reading the record first gives you the chance to check its contents before updating it.
line 11	Close the ISAM file.

Deleting records

Deleting a record is similar to reading a record:

1. Open the ISAM file.
2. Fill the buffer's index field with a search value.
3. Call `isdelete` to delete the matching record.

This program fragment illustrates these steps:

```
1 int fdbook ;
2 char buffer[RECSIZ] ;
3 long isbn ;
4 fdbook = isopen("books", ISINOUT + ISEXCLLOCK) ;
5 isbn = 0810462613 ;
6 stlong(isbn, buffer) ;
7 isdelete(fdbook, buffer) ;
8 isclose(fdbook) ;
```

- | | |
|-----------|---|
| lines 1-4 | Declare file descriptor, buffer to hold selected record, and index key; open the file and search for the appropriate record as in previous examples. |
| line 6 | Insert search value into buffer. |
| line 7 | Delete the record that matches. Note that the <code>isdelete</code> function does not itself provide an opportunity for confirming the deletion. In an interactive application, you may want to use the <code>isdelcurr</code> function instead. That function deletes the most recently read record, which gives you a chance to display the contents and obtain confirmation. |

Adding an index

Although you must specify a primary index when creating an ISAM file, you may want to add secondary indexes to access your data by different fields.

Each index that you add is updated each time a record is added, updated, or deleted to the file. Too many indexes can slow processing speeds. To reduce this overhead, you may want to keep only a single primary index and create temporary indexes as needed. The trade-off is the time required to build the temporary index, which can be considerable for a file that contains a large number of data records.

These steps create a secondary index:

1. Open the ISAM file in exclusive mode.
2. Fill a **keydesc** structure with values.
3. Call **isaddindex** to create the index.

The following program fragment illustrates these steps by adding an index on the "Title" field.

```

1 struct keydesc key ;
2 int fdbook ;
3 fdbook = isopen("books", ISINOUT + ISEXCLLOCK) ;
4 key.k_flags = ISDUPS ;
5 key.k_nparts = 1 ;
6 key.kpart[0].kp_start = 4 ;
7 key.kpart[0].kp_leng = 20 ;
8 key.kpart[0].kp_type = CHARTYPE ;
9 isaddindex (fdbook, &key) ;
10 isdelindex (fdbook, &key) ;
11 isclose(fdbook) ;

```

line 1	Index key structure
line 2-3	Declare file descriptor and open the ISAM file as in previous examples.
line 4	Duplicate values are allowed.
line 5	Number of key parts
line 6	Position of first byte
line 7	Length of key part
line 8	Data type of key part
line 9	Create the new index. The new index created with isaddindex becomes the current index for the open file. To switch to another index, use the isstart function.
line 10	Delete the index
line 11	Close the ISAM file

Chapter 4

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-code 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 and the articles entitled "Message catalog", "Internationalization", and "Localization" in the *Encyclopedia*.

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_ALL`, `LANG`, and `NLSPATH`. The precise meanings of these variables are defined in the "locale", "LANG", and "NLSPATH" articles of the *Encyclopedia* and the `setlocale(S)` manual page.

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 4-1 English language message catalog (english/hello.text)

```
1 $set 1
2 1 Hello everyone. It's
```

- 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.

Example 4-2 French language message catalog (french/hello.text)

```
1 $set 1
2 1 Bonjour tout le monde. C'est
```

- line 1-2 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 the section before. Rather than hard-coding output directly, it reads the proper message catalog depending on the LANG environment variable.

Example 4-3 Internationalized "Hello, world" program (hello.c)

```

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

5 extern char *catgets();

6 main()
7 {
8     nl_catd catalog;
9     long    clock;

10    /* This call is not needed under SCO Unix, it is done from crt0
11     *
12     * setlocale (LC_ALL, "");
13     */

14    if ((catalog = catopen ("hello", 0)) == -1) {
15        fprintf(stderr, "Can't open message catalog /
16        /* Continue anyway, this is not fatal */
17    }

18    (void) time(&clock);

19    printf("%s %s /0,
20        catgets(catalog, 1, 1, "hello world (English)"),
21        nl_cxtime(&clock, "%c")
22    );

23    catclose(catalog);

24    exit(0);
25 }
```

line 1	Needs to be included to use standard routines like printf(S) .
line 2	Needs to be included to use the catalog routines

lines 3-4	Needs to be included to use the <code>LC_*</code> and <code>LANG</code> environment variables
line 5	Defines the catalog accessing function
line 8	Defines a variable to be of a type that identifies a catalog (referred to as a catalog descriptor) .
lines 10-13	As the comment says, this line is not necessary for SCO UNIX but would need to be included on other systems so that the local environment variable <code>LANG</code> is read.
line 14	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 <code>catalog</code> variable.
lines 15-17	Prints out error message upon failure.
line 18	Sets the <code>clock</code> variable to UNIX operating system Epoch time.
line 19	Describes the output format string.
line 20	Retrieves message number 1 of set number 1 in the message catalog designated by the variable <code>catalog</code> variable. The final string is a default if there is an error in reading the message catalog.
line 21	Prints the time according to the format specified by the <code>nl_cxtime(S)</code> routine.
line 23	Closes the message catalog identified by the variable <code>catalog</code> .

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 4-4 Setup program

```

1      set -x
2      NLSPATH=./%l/%N.cat
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 <code>NLSPATH</code> variable which contains the path where the message catalogs are contained. The <code>"%l"</code> extracts the string defined in the <code>LANG</code> environment variable and the <code>"%N"</code> extracts the string from the <i>name</i> argument from the <code>catopen(S)</code> routine. If the value in the <code>LANG</code> variable is "english", the message catalog file in our example would be <code>./english/hello.cat</code> |
| line 3 | Exports this definition so that the <code>NLSPATH</code> variable is defined permanently in this shell session. |
| lines 4-5 | Creates the <code>.cat</code> 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 <code>LANG</code> 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 <code>LANG</code> 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 XPG3 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 conform to the XPG3 format (see the "locale" article in the *Encyclopedia* for this formatting scheme).

Chapter 5

OMF/x.out format

This chapter presents the object record formats that define the relocatable object language for the iAPX-86 family of microprocessors. The 8086 object language is the output of all language translators that have the 8086 as their target processor and are linked by the link editor. The 8086 object language is input and output for object language processors such as linkers and librarians.

NOTE This chapter is provided as a convenience; for more information on this subject contact Intel Corporation and Microsoft, Incorporated.

Except where otherwise noted, references to the 8086 in this document refer to the 8086/80286/80386 processors. In general, the 8086/80286 references are made to 16-bit offsets and 64K segment offsets, which do not apply to the 80386.

Also note that explanations of common OMF terms are presented at the end of this chapter.

The 8086 object module formats permit you to specify relocatable memory images that may be linked together. The formats allow efficient use of the memory-mapping facilities of the 8086 microprocessor.

The following record formats, as described in this chapter, are supported. Those formats preceded by an asterisk (*) deviate from the Intel specification.

Object Module Record Formats

T-Module Header Record
 List of Names Record
 *Segment Definition Record
 *Group Definition Record
 *Type Definition Record
 *Public Names Definition Record
 *External Names Definition Record
 *Line Numbers Record
 Logical Enumerated Data Record
 Logical Iterated Data Record
 Fixup Record
 *Module End Record
 Comment Record

OMF Records

An object code file must contain a sequence of one or more modules or a library containing zero or more modules. A module is defined as a collection of object code defined by a sequence of object records.

A module is a collection of object code defined by a sequence of records produced by a translator. The object code represents contiguous regions of memory whose contents are determined at translation time. These regions are called logical segments (LSEGs). A module defines the attributes of each LSEG. The segment definition record (SEGDEF) is the vehicle by which all LSEG information (name, length, memory alignment, and so on) is maintained. The LSEG information is required when multiple LSEGs are combined and when segment addressability is established. The SEGDEF records must follow the first header record.

The 8086/80286 addressing mechanism provides segment base registers from which a 64-Kbyte region of memory, called a frame, may be addressed. There are one code-segment base register (CS), two data-segment base registers (DS, ES), and one stack-segment base register (SS). The 80386 has two additional segment registers: FS and GS, and can address up to (2^{32}) bytes of memory from each segment register.

The greatest possible number of LSEGs that may make up a memory image far exceeds the number of available base registers. Thus, base registers may require frequent loading. This would occur in a modular program with many small data and/or code LSEGs.

Since such frequent loading of base registers is undesirable, it is a good strategy to collect many small LSEGs together into a single unit that fits in one memory frame so that all the LSEGs may be addressed using the same base register value. This addressable unit is a group.

A module header record is always the first record in a module and provides the module name.

In addition to a name, a module may have the attribute of being a main program and may have a specified starting address. When you are linking multiple modules together, only one module with the main attribute should be given.

In summary, modules may or may not be main and may or may not have a starting address.

Record order

The following syntax shows the valid orderings of records to form a module. In addition, the given semantic rules provide information about how to interpret the record sequence.

NOTE The syntactic description language used below is defined in Wirth: CACM, November 1977, vol.#20, no.#11, pages 822-823. The character strings represented by capital letters are not literals, but are identifiers that are further defined in the section describing the record formats.

Table 5-1 Record order

object file	= tmodule
tmodule	= THEADR seg-grp {component} modtail
seg_grp	= {LNAMES} {SEGDEF} {TYPDEF EXTDEF GRPDEF}
component	= data debug_record
data	= content_def thread_def TYPDEF PUBDEF EXTDEF
debug_record	= LINNUM
content_def	= data_record {FIXUPP}
thread_def	= FIXUPP (containing only thread fields)
data_record	= LIDATA LEDATA
modtail	= MODEND

The following rules apply:

- A FIXUPP record always refers to the previous data record.

- All LNames, SEGDEF, GRPDEF, TYPDEF, and EXTDEF records must precede all records that refer to them.
- Comment records may appear anywhere in a file, except as the first or last record in a file or module, or within a content_def.

Numeric list of record types

Table 5-2 Numeric list of record types

*6E	RHEADR	*92	LOCSYM
*70	REGINT	*93	MLOC386
*72	REDATA	94	LINNUM
*74	RIDATA	95	MLIN386
*76	OVLDEF	96	LNames
*78	ENDREC	98	SEGDEF
*7A	BLKDEF	99	MSEG386
*7C	BLKEND	9A	GRPDEF
*7E	DEBSYM	9C	FIXUPP
80	THEADR	9D	MFIX386
*82	LHEADR	*9E	(none)
*84	PEDATA	A0	LEDATA
*86	PIDATA	A1	MLED386
88	COMENT	A2	LIDATA
8A	MODEND	A3	MLID386
8B	H386END	*A4	LIBHED
8C	EXTDEF	*A6	LIBNAM
8E	TYPDEF	*A8	LIBLOC
90	PUBDEF	*AA	LIBDIC
91	MPUB386		

NOTE The record types marked with an asterisk are not supported by the link editor. They will be ignored if they are found in an object module.

Introduction to the record formats

The following pages present diagrams of record formats in schematic form. Here is a sample record format, to illustrate the various conventions.

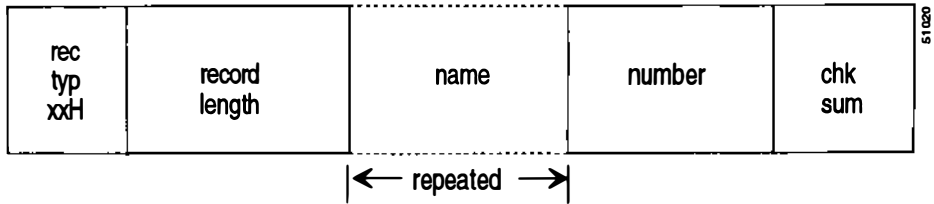


Figure 5-1 Sample record format (samrec)

Title and official abbreviation

At the top is the name of the record format described, with an official abbreviation. To promote uniformity among various programs, including translators and debuggers, the abbreviation should be used in both code and documentation. The record format abbreviation is always six letters.

The boxes

Each format is drawn with boxes of two sizes. The narrow boxes represent single bytes. The wide boxes represent two bytes each. The wide dashed boxes represent a variable number of bytes, one or more, depending upon content. The wide solid boxes represent 4-byte fields.

Rectyp

The first byte in each record contains a value between 0 and 255, indicating the record type. For records that have both 16- and 32-bit versions, the low-order bit of the record type indicates the type: 0=16-bit, 1=32 bit.

Record length

The second field in each record contains the number of bytes in the record, exclusive of the first two fields.

Name

Any field that indicates a "name" has the following internal structure: the first byte contains a number between 0 and 127, inclusive, that indicates the number of remaining bytes in the field. The remaining bytes are interpreted as a byte string.

Most translators constrain the character set to be a subset of the ASCII character set.

Number

A 4-byte number field represents a 32-bit unsigned integer, where the first 8 bits (least-significant) are stored in the first byte (lowest address), the next 8 bits are stored in the second byte, and so on.

Repeated or conditional fields

Some portions of a record format contain a field or a series of fields that may be repeated one or more times. Such portions are indicated by the "repeated" or "rpt" brackets below the boxes.

Similarly, some portions of a record format are present only if some given condition is true; these fields are indicated by similar "conditional" or "cond" brackets below the boxes.

Checksum

The last field in each record is a check sum, which contains the 2's complement of the sum (modulo 256) of all other bytes in the record. Therefore, the sum (modulo 256) of all bytes in the record equals 0.

Bit fields

Descriptions of contents of fields will sometimes be at the bit level. Boxes with complete vertical lines drawn through them represent bytes or words; the partial vertical lines indicate bit boundaries; thus the byte represented below, has three bit-fields of 3-, 1-, and 4-bits.

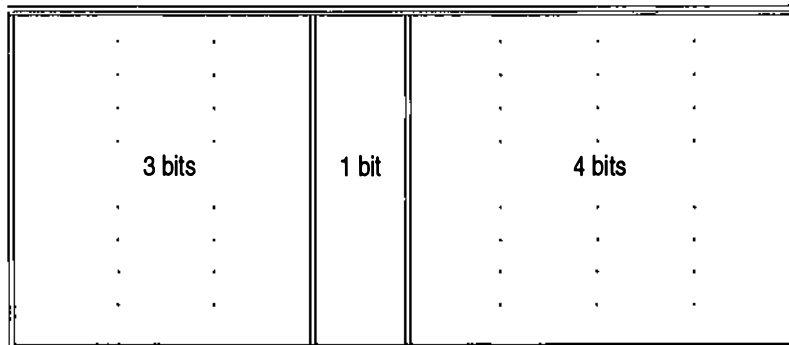


Figure 5-2 Bit field

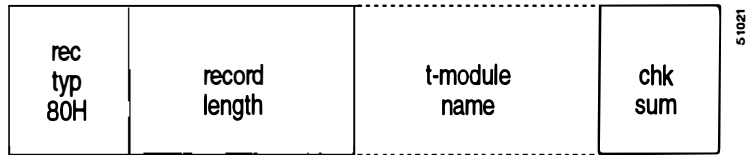


Figure 5-3 T-module Header Record (THEADR)

Every module output from a translator must have a T-module HEADER record.

T-module name

The T-module name provides a name for the T-module.

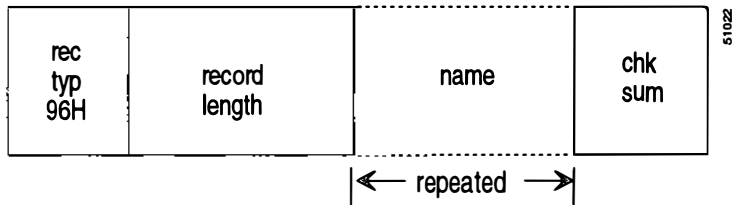


Figure 5-4 List of names record (LNAMES)

This record provides a list of names that may be used in following SEGDEF and GRPDEF records as the names of Segments, Classes, and/or Groups.

The ordering of LNAMES records within a module, together with the ordering of names within each LNAMES record, induces an ordering on the names. Thus, these names are considered to be numbered: 1, 2, 3, 4, ... These numbers are used as "Name Indices" in the segment name index, class name index, and group name index fields of the SEGDEF and GRPDEF records.

Name

Name is a repeatable field which provides a name and which may have zero length.

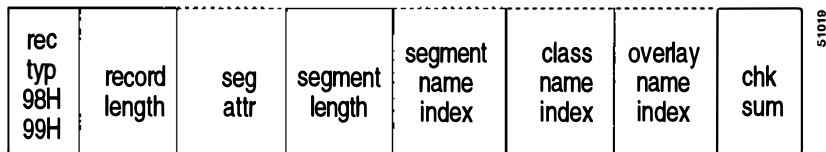


Figure 5-5 Segment definition record (SEGDEF)

Segment index values 1 through 32767, which are used in other record types to refer to specific LSEGs, are defined implicitly by the sequence in which SEGDEF records appear in the object file.

In the `rec typ` field, 98H and 99H describe 16- and 32-bit segments, respectively.

Seg attr

The `seg attr` field provides information on various attributes of a segment, and has the following format:

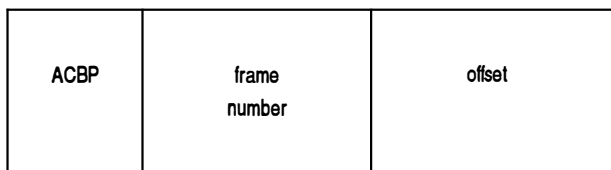
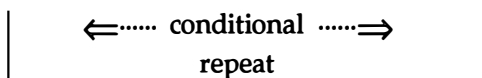


Figure 5-6 Seg attr field



The ACBP byte contains four numbers which are the A, C, B, and P attribute specifications. This byte has the following format:

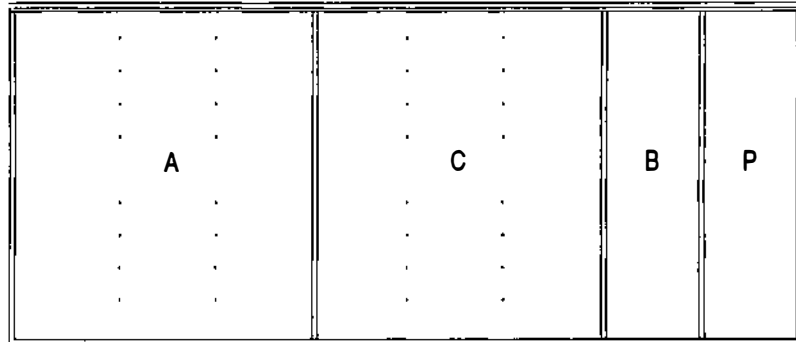


Figure 5-7 ACBP byte

"A" (Alignment) is a 3-bit subfield that specifies the alignment attribute of the LSEG. The semantics are defined as follows:

- A=0 SEGDEF describes an absolute LSEG.
- A=1 SEGDEF describes a relocatable, byte-aligned LSEG.
- A=2 SEGDEF describes a relocatable, word-aligned LSEG.
- A=3 SEGDEF describes a relocatable, paragraph-aligned LSEG.
- A=4 SEGDEF describes a relocatable, page-aligned LSEG.
- A=5 SEGDEF describes a relocatable, double-word-aligned LSEG.
(386 OMF only)

If A=0, the frame number and offset fields will be present. Using `ld`, absolute segments may be used for addressing purposes only. For example, the starting address of a ROM and the symbolic names for addresses within the ROM may be defined in this way. `ld` will ignore any data specified as belonging to an absolute LSEG.

"C" (Combination) is a 3-bit subfield that specifies the combination attribute of the LSEG. Absolute segments (A=0) must have combination zero (C=0). For relocatable segments, the C field encodes a number (0,1,2,4,5,6 or 7) that indicates how the segment can be combined. The interpretation of this attribute is best given by considering how two LSEGs are combined:

- Let X,Y be LSEGs, and let Z be the LSEG resulting from the combination of X,Y.
- Let LX and LY be the lengths of X and Y, and let MXY denote the maximum of LX, LY.

- Let G be the length of any gap required between the X- and Y-components of Z to accommodate the alignment attribute of Y.
- Let LZ denote the length of the (combined) LSEG Z; let dx ($0 \leq dx < LX$) be the offset in X of the (combined) LSEG Z; let dy ($0 \leq dy < LY$) be the offset in Y of a byte, and let dy similarly be the offset in Y of a byte.

The following table gives the length LZ of the combined LSEG Z, and the offsets dx' and dy' in Z for the bytes corresponding to dx in X and dy in Y. Intel additionally defines alignment types 5 and 6 and also processes code and data placed in segment with align-type.

Table 5-3 Combination attribute table

C	LZ	dx'	dy'	
2	LX+LY+G	dx	dy+LX+G	Public
5	LX+LY+G	dx	dy+LX+G	Stack
6	MAX	dx	dy	Common

The table has no lines for C=0, C=1, C=3, C=4, and C=7. C=0 indicates that the relocatable LSEG may not be combined; C=1 and C=3 are undefined. C=4 and C=7 are treated like C=2. C1, C4, and C7 all have different meanings according to the Intel standard.

"B" (Big) is a 1-bit subfield which, if 1, indicates that the segment length is exactly 2^{16} (2^{32} in the case of 32-bit segments). In this case the segment length field must contain zero.

The "P" field must always be zero. The "P" field is the "Page resident" field according to the Intel standard.

The frame number and offset fields (present only for absolute segments, A=0) specify the placement in MAS of the absolute segment. The range of offset is constrained to be between 0 and 15 inclusive. If a value larger than 15 is desired for offset, then an adjustment of the frame number should be done.

Segment length

The segment length field gives the length of the segment in bytes. The length may be zero; if so, ld will *not* delete the segment from the module. The segment length field is two bytes for a 16-bit segment (Rectyp 98) and four bytes for a 32-bit segment (Rectyp 99). This is large enough for numbers up to $(2^{16})-1$ and $(2^{32})-1$, respectively. The B attribute bit in the ACBP field (see SEG ATTR section) must be used to indicate a length of (2^{16}) or (2^{32}) .

Segment name index

The segment name is a name the programmer or translator assigns to the segment. Examples: CODE, DATA, MODULENAME_CODE, STACK, TAXDATA. This field provides the Segment Name, by indexing into the list of names provided by the L NAMES record(s).

Class name index

The class name is a name the programmer or translator can assign to a segment. If none is assigned, the name is null, and has length 0. The purpose of class names is to allow the programmer to define a “handle” used in the ordering of the LSEGS in MAS. Examples: RED, WHITE, BLUE; ROM FASTRAM, DISPLAYRAM. This field provides the class name, by indexing into the list of names provided by the L NAMES record(s).

Overlay name index

NOTE This is ignored in ld versions 2.40 and later, but supported in all earlier versions. However, semantics differ from Intel semantics.

The overlay name is a name the translator and/or ld, at the programmer’s request, applies to a segment. The overlay name, like the class name, may be null. This field provides the overlay name, by indexing into the list of names provided by the L NAMES record(s).

NOTE The “complete name” of a segment is a 3-component entity comprising a segment name, a class name, and an overlay name. (The latter two components may be null.)

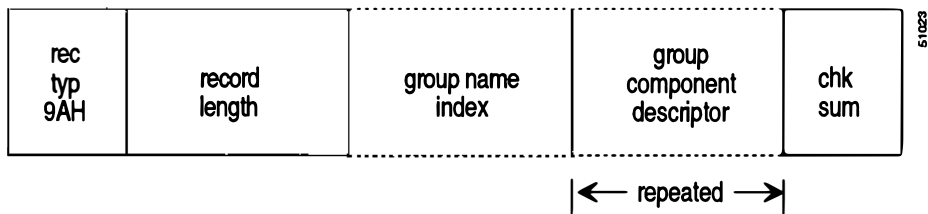


Figure 5-8 Group definition record (GRPDEF)

Group name index

The group name is a name by which a collection of LSEGs may be referenced. The important property of such a group is that, when the LSEGs are eventually fixed in MAS, there must exist some frame which “covers” every LSEG of the group.

The group name index field provides the group name, by indexing into the list of names provided by the LNAMEs record(s).

Group component descriptor

Each group component descriptor has the following format:

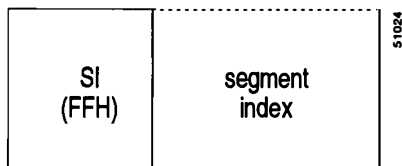


Figure 5-9 Group component descriptor

The first byte of the descriptor contains 0FFH; the descriptor contains one field, which is a segment index that selects the LSEG described by a preceding SEGDEF record.

Intel defines four other group descriptor types, each with its own meaning. They are 0FEH, 0FDH, 0FBH, and 0FAH. The link editor will treat all of these values the same as 0FFH (that is, it always expects 0FFH followed by a segment index, and it does not check to see if the value is actually 0FF).

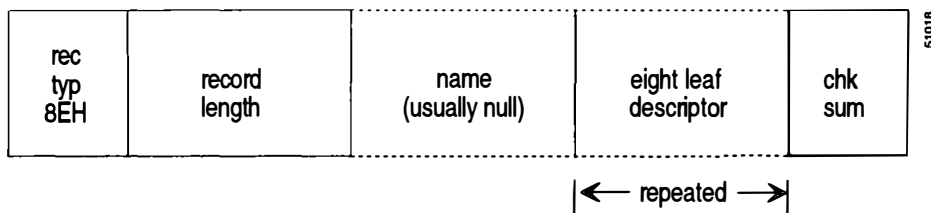


Figure 5-10 Type definition record

The link editor uses TYPDEF records only for communal variable allocation. This is *not* Intel's intended purpose. See "Type Representations for Communal Variables."

As many eight-leaf descriptor fields as necessary are used to describe a branch. (Every such field except the last in the record describes eight leaves; the last such field describes from one to eight leaves.)

Type index values 1 through 32767, which are contained in other record types to associate object types with object names, are defined implicitly by the sequence in which TYPDEF records appear in the object file.

Name

Use of this field is reserved. Translators should place a single byte containing 0 in it (the representation of a name of length zero).

Eight-leaf descriptor

This field can describe up to eight leaves.

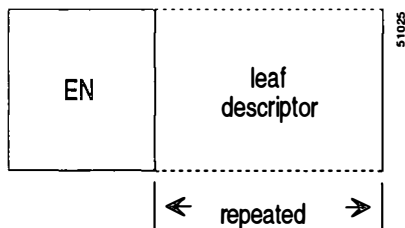


Figure 5-11 Eight-leaf descriptor

The EN field is a byte: the 8 bits, left to right, indicate if the following 8 leaves (left to right) are Easy (bit=0) or Nice (bit=1).

The leaf descriptor field, which occurs between 1 and 8 times, has one of the following formats:

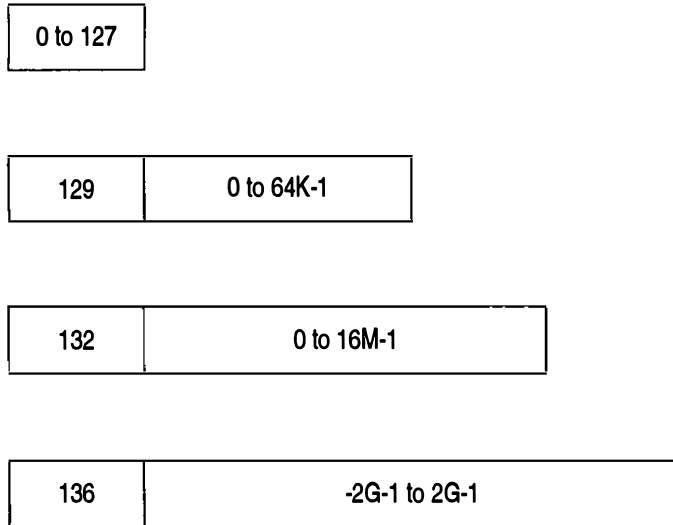


Figure 5-12 Leaf descriptor field

The first format (single byte), containing a value between 0 and 127, represents a numeric leaf whose value is the number given.

The second format, with a leading byte containing 129, represents a numeric leaf. The number is contained in the following two bytes.

The third format, with a leading byte containing 132, represents a numeric leaf. The number is contained in the following three bytes.

The fourth format, with a leading byte containing 136, represents a signed numeric leaf. The number is contained in the following four bytes, sign extended if necessary.

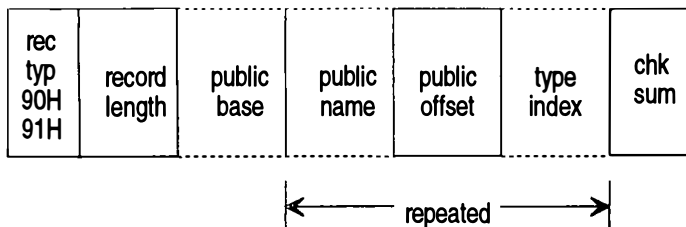


Figure 5-13 Public names definition record (PUBDEF)

This record provides a list of one or more public names; for each one, three data are provided: (1) a base value for the name, (2) the offset value of the name, and (3) the type of entity represented by the name.

In the record type field, 90H and 91H describe 16- and 32-bit public definition records, respectively.

Public base

The public base has the following format:

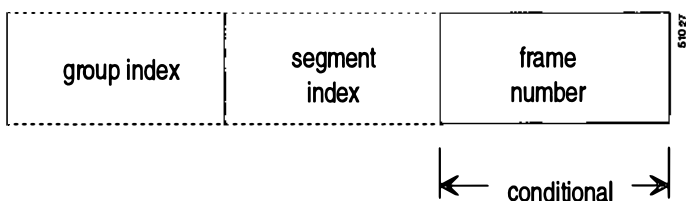


Figure 5-14 Public base

The group index field has a format given earlier, and provides a number between 0 and 32767 inclusive. A non-zero group index associates a group with the public symbol, and is used as described in the section of this chapter titled "Conceptual Framework for Fixups," case (F2c). A zero group index indicates that there is no associated group.

The segment index field has a format given earlier, and provides a number between 0 and 32767, inclusive.

A non-zero segment index selects an LSEG. In this case, the location of each public symbol defined in the record is taken as a non-negative displacement (given by a public offset field) from the first byte of the selected LSEG, and the frame number field must be absent.

A segment index of 0 (legal only if group index is also 0) means that the location of each public symbol defined in the record is taken as a displacement from the base of the frame defined by the value in the frame number field.

The frame number is present if both the segment index and group index are zero.

A non-zero group index selects some group; this group is taken as the “frame of reference” for references to all public symbols defined in this record; that is, `ld` will perform the following:

1. Any fixup of the form:

```
TARGET: EI(P)
FRAME: TARGET
```

(where “P” is a public symbol in this PUBDEF record) will be converted by `ld` to a fixup of the form:

```
TARGET: SI(L),d
FRAME: GI(G)
```

where “SI(L)” and “d” are provided by the segment index and public offset fields. (The “normal” action would have the frame specifier in the new fixup be the same as in the old fixup: `FRAME: TARGET`.)

2. When the value of a public symbol, as defined by the segment index, public offset, and (optional) frame number fields, is converted to a {base,offset} pair, the base part will be taken as the base of the indicated group. If a non-negative 16-bit offset cannot then complete the definition of the public symbol’s value, an error occurs.

A group index of zero selects no group. `ld` will not alter the frame specification of fixups referencing the symbol, and will take, as the base part of the absolute value of the public symbol, the canonic frame of the segment (either LSEG or PSEG) determined by the segment index field.

Public name

The public name field gives the name of the object whose location in MAS is made available to other modules. The name must contain one or more characters.

Public offset

The public offset field is a 16-bit value (Rectyp=90H), or a 32-bit value (Rectyp=91H), which is either the offset of the public symbol with respect to an LSEG (if segment index > 0), or the offset of the public symbol with respect to the specified frame (if segment index = 0).

Type index

The type index field identifies a single preceding TYPDEF (type definition) record containing a descriptor for the type of entity represented by the public symbol. This field is ignored by ld.

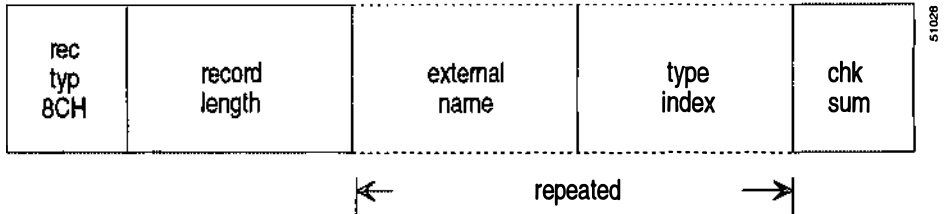


Figure 5-15 External names definition record (EXTDEF)

This record provides a list of external names, and for each name, the type of object it represents. ld will assign to each external name the value provided by an identical public name (if such a name is found).

External name

This field provides the name, which must have non-zero length, of an external object.

Inclusion of a name in an external names record is an implicit request that the object file be linked to a module containing the same name declared as a public symbol. This request obtains whether or not the external name is referenced within some FIXUPP record in the module.

The ordering of EXTDEF records within a module, together with the ordering of external names within each EXTDEF record, induces an ordering on the set of all external names requested by the module. Thus, external names are considered to be numbered 1, 2, 3, 4, These numbers are used as external indices in the target datum and/or frame datum fields of FIXUPP records to refer to a particular external name.

NOTE 8086 external names are numbered positively: 1,2,3,... This is a change from 8080 external names, which were numbered starting from zero: 0,1,2,... This conforms with other 8086 indices (segment index, type index, and so on) which use 0 as a default value with special meaning.

Type index

This field identifies a single preceding TYPDEF (type definition) record containing a descriptor for the type of object named by the external symbol.

The type index is used only in communal variable allocation by the link editor.

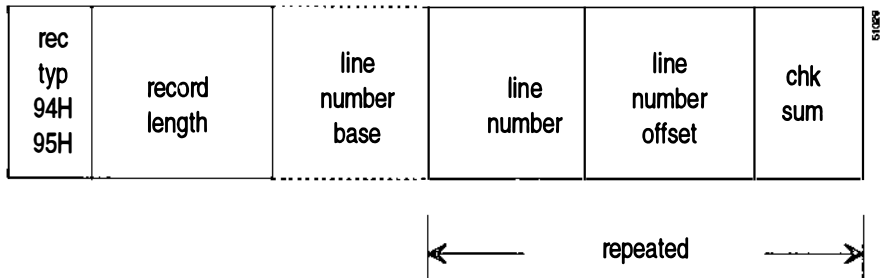


Figure 5-16 Line numbers record (LINNUM)

This record provides the means by which a translator may pass the correspondence between a line number in source code and the corresponding translated code.

In the record type field, 94H and 95H describe 16- and 32-bit line number records, respectively.

Line-number base

The line-number base has the following format:

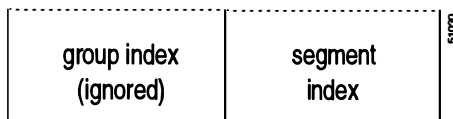


Figure 5-17 Line-number base

The segment index determines the location of the first byte of code corresponding to some source line number.

Line-number

A line number between 0 and 32767, inclusive, is provided in binary by this field. The high-order bit is reserved for future use and must be zero.

Line-number offset

The line-number offset field is either a 16-bit value (Rectyp=94H) or a 32-bit value (Rectyp=95H) that is the offset of the line number with respect to an LSEG (if segment index > 0).

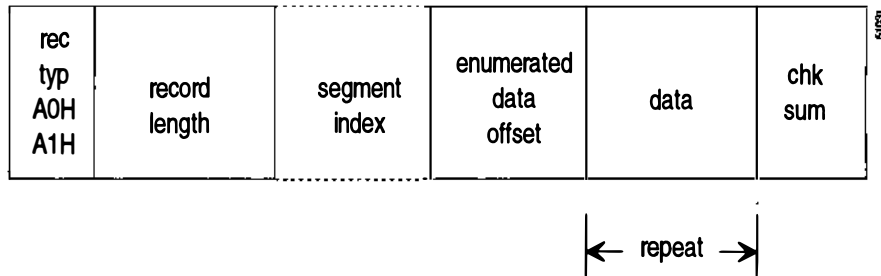


Figure 5-18 Logical enumerated data record (LEDATA)

This record provides contiguous data from which a portion of an 8086 memory image may be constructed.

In the record type field, A0H and A1H describe 16- and 32-bit LEDATA records, respectively.

Segment index

This field must be non-zero and specifies an index relative to the segment definition records found previous to the LEDATA record.

Enumerated data offset

This field specifies either a 16-bit offset (Rectype=A0H) or a 32-bit offset (Rectype=A1H) that is relative to the base of the LSEG specified by the segment index and defines the relative location of the first byte of the DAT field. Successive data bytes in the DAT field occupy successively higher locations of memory.

Data

This field provides up to 1024 consecutive bytes of relocatable or absolute data.

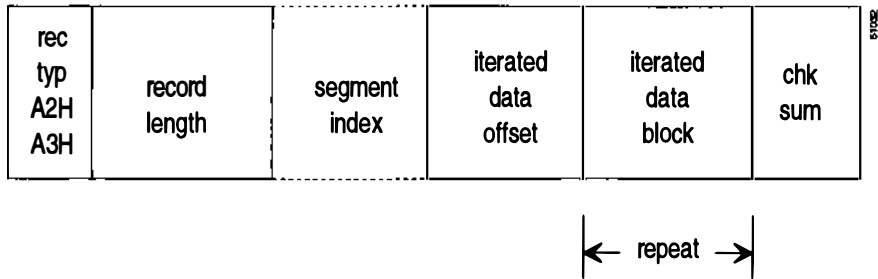


Figure 5-19 Logical iterated data record (LIDATA)

This record provides contiguous data from which a portion of an 8086 memory image may be constructed.

In the record type field, A2H and A3H describe 16- and 32-bit LIDATA records, respectively.

Segment index

This field must be non-zero and specifies an index relative to the SEGDEF records found previous to the LIDATA record.

Iterated data offset

This field specifies either a 16-bit offset (Rectype=A2H) or a 32-bit offset (Rectype=A3H) that is relative to the base of the LSEG specified by the segment index and defines the relative location of the first byte in the iterated data block. Successive data bytes in the iterated data block occupy successively higher locations of memory.

Iterated data block

This repeated field is a structure specifying the repeated data bytes. The structure has the following format:

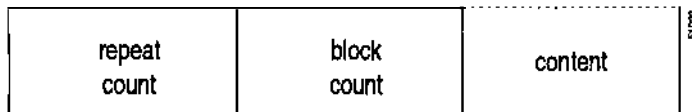


Figure 5-20 Iterated data block

NOTE The link editor cannot handle LIDATA records whose iterated data block is larger than 512 bytes.

Repeat count

This field specifies the number of times that the content portion of this iterated data block is to be repeated. repeat count must be non-zero.

Block count

This field specifies the number of iterated data blocks that are to be found in the content portion of this iterated data block. If this field has value zero, then the content portion of this iterated data block is interpreted as data bytes. If non-zero, then the content portion is interpreted as that number of iterated data blocks.

Content

This field may be interpreted in one of two ways, depending on the value of the previous block count field.

If block count is zero, then this field is a 1-byte count followed by the indicated number of data bytes.

If block count is non-zero, then this field is interpreted as the first byte of another iterated data block.

NOTE From the outermost level, the number of nested iterated data blocks is limited to 17, i.e., the number of levels of recursion is limited to 17.

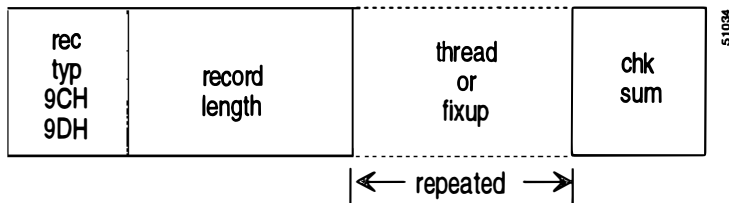


Figure 5-21 Fixup record

This record specifies 0 or more fixups. Each fixup requests a modification (fixup) to a location within the previous data record. A data record may be followed by more than one fixup record that refers back to it. Each fixup is specified by a fixup field that specifies four data: a location, a mode, a target, and a frame. The frame and the target may be specified totally within the fixup field, or may be specified by reference to a preceding thread field.

A thread field specifies a default target or frame that may subsequently be referred to in identifying a target or a frame. Eight threads are provided: four for frame specification and four for target specification. Once a target or frame has been specified by a thread, it may be referred to by following fixup

fields (in the same or following FDXUPP records), until another thread field with the same type (target or frame) and thread number (0 - 3) appears (in the same or another FDXUPP record).

In the record type field, 9CH and 9DH describe 16- and 32-bit FDXUPP records, respectively.

Thread

Thread is a field with the following format:

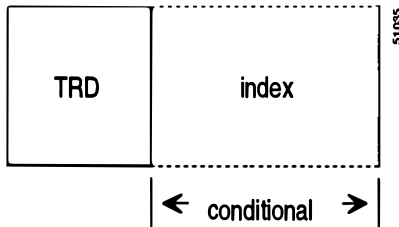


Figure 5-22 Thread

The TRDDAT (ThRead DATA) subfield is a byte with this internal structure:

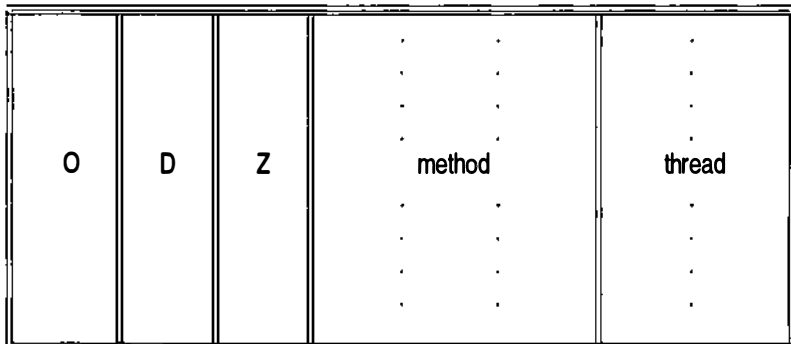


Figure 5-23 Thread data subfield

The "Z" is a 1-bit subfield, currently without any defined function, that is required to contain 0.

The "D" subfield is one bit that identifies what type of thread is being specified. If D=0, then a target thread is being defined; if D=1, then a frame thread is being defined.

Method is a 3-bit subfield containing a number between 0 and 3 (D=0) or a number between 0 and 6 (D=1).

If $D=0$, then $\text{method} = (0, 1, 2, 3, 4, 5, 6, 7) \bmod 4$, where the 0, ..., 7 indicate methods T0, ..., T7 of specifying a target. Thus, method indicates what kind of index or frame number is required to specify the target, without indicating if the target will be specified in a primary or secondary way. Note that methods 2b, 3, and 7 are not supported by ld.

If $D=1$, then $\text{method} = 0, 1, 2, 4, 5$, corresponding to methods F0, ..., of specifying a frame. Here, method indicates what kind (if any) of index is required to specify the frame. Note that methods 3 and 5d are not supported by ld.

Thread is a number between 0 and 3, and associates a thread Number to the frame or target defined by the thread field.

Index contains a segment index, group index, or external index depending on the specification in the method subfield of the TRD field. This subfield will not be present if F4 or F5 are specified by method.

Fixup

FDXUP is a field with the following format:

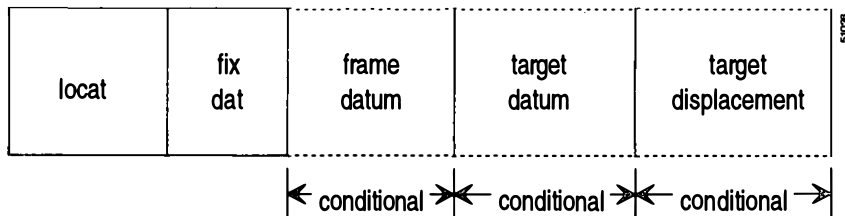


Figure 5-24 Fixup field

LOCAT is a byte pair with the following format:

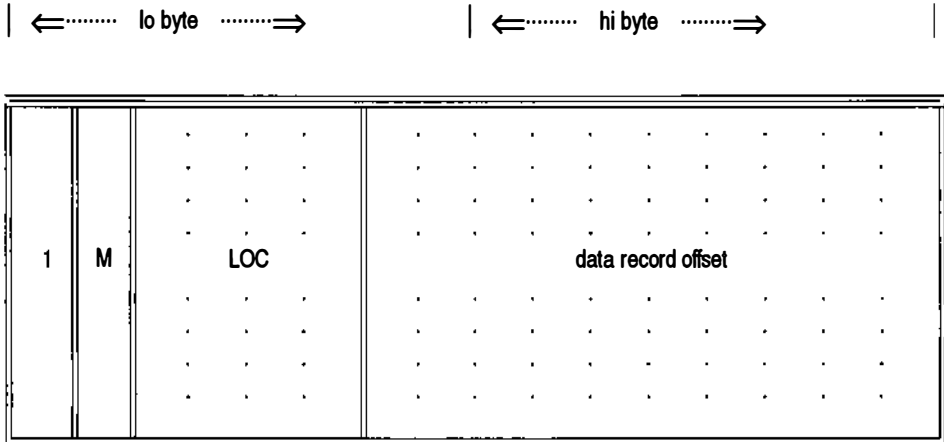


Figure 5-25 Locat

M is a 1-bit subfield that specifies the mode of the fixups: self-relative (M=0) or segment-relative (M=1).

NOTE Self-relative fixups may not be applied to LIDATA records.

LOC is a four-bit subfield indicating the type of location that is to be fixed up:

0	-	8 bit	lo byte
1	-	16 bit	offset
2	-	16 bit	base
3	-	32 bit	pointer
4	-	8 bit	hi byte
5	-	16 bit	offset (linker resolved)
9	-	32 bit	offset
11	-	48 bit	pointer
13	-	32 bit	offset (linker resolved)

LOC values 9, 11, and 13 are only valid in 32-bit FIXUPP records (record type 9D). All values not mentioned are invalid.

The data record offset is a number between 0 and 1023, inclusive, that gives the relative position of the lowest order byte of location (the actual bytes being fixed up) within the preceding data record. The data record offset is relative to the first byte in the data fields in the data records.

NOTE It is possible for the value of data record offset to designate a “location” within a repeat count subfield or a block count subfield of the iterated data field. Such a reference is an error. The action of ld on such a malformed record is undefined.

FIX DAT is a byte with the following format:

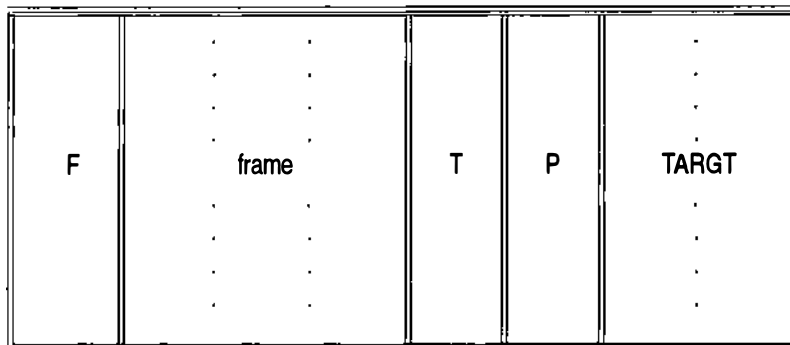


Figure 5-26 Fix dat

NOTE Frame method 2b, F3, and F5d are not supported. Target method T3 and T7 are not supported.

F is a 1-bit subfield that specifies whether the frame for this FIXUP is specified by a thread (F=1) or explicitly (F=0).

frame is a number interpreted in one of two ways as indicated by the F bit. If F is zero, frame is a number between 0 and 5 and corresponds to methods F0, ..., F5 of specifying a frame. If F=1, then frame is a thread number (0-3). It specifies the frame most recently defined by a thread field that defined a frame thread with the same thread number. (Note that the thread field may appear in the same, or in an earlier FIXUPP record.)

“T” is a 1-bit subfield that specifies whether the target specified for this fixup is defined by reference to a thread (T=1), or is given explicitly in the FIXUP field (T=0).

“P” is a 1-bit subfield that indicates whether the target is specified in a primary way (requires a target displacement, P=0) or specified in a secondary way (requires no target displacement, P=1). Since a target thread does not have a primary/secondary attribute, the P bit is the only field that specifies the primary/secondary attribute of the target specification.

TARGT is interpreted as a 2-bit subfield. When T=0, it provides a number between 0 and 3, corresponding to methods T0, ..., T3 or T4, ..., T7, depending on the value of P (P can be interpreted as the high-order bit of T0, ..., T7).

When the target is specified by a thread (T=1), then TARGT specifies a thread number (0-3).

Frame datum is the “referent” portion of a frame specification, and is a segment index, a group index, or an external index. The frame datum subfield is present only when the frame is specified neither by a thread (F=0) nor explicitly by methods F4 or F5 or F6.

Target datum is the “referent” portion of a target specification, and is a segment index, a group index, an external index, or a frame number. The target datum subfield is present only when the target is not specified by a thread (T=0).

Target displacement is the displacement required by “primary” methods of specifying targets. This field is 2 bytes long in 16-bit FIXUPP records (Rectyp=9CH) and 4 bytes long in 32-bit FIXUPP records (Rectyp=9DH). This subfield is present if P=0.

NOTE All these methods are described in the section of this chapter titled “Conceptual Framework for Fixups.”

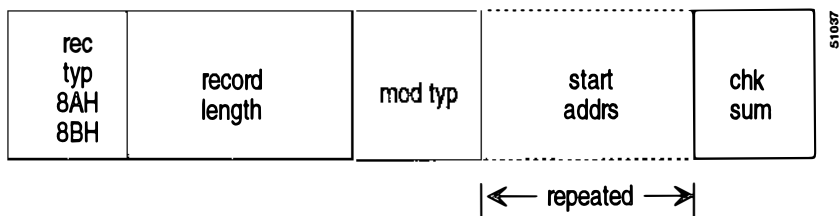


Figure 5-27 Module end record (MODEND)

This record serves two purposes. It denotes the end of a module and indicates whether the module just terminated has a specified entry point for initiation of execution. If the latter is true, the execution address is specified.

In the record type field, 8AH and 8BH describe 16- and 32-bit MODEND records, respectively.

Mod type

This field specifies the attributes of the module. The bit allocation and associated meanings are as follows:

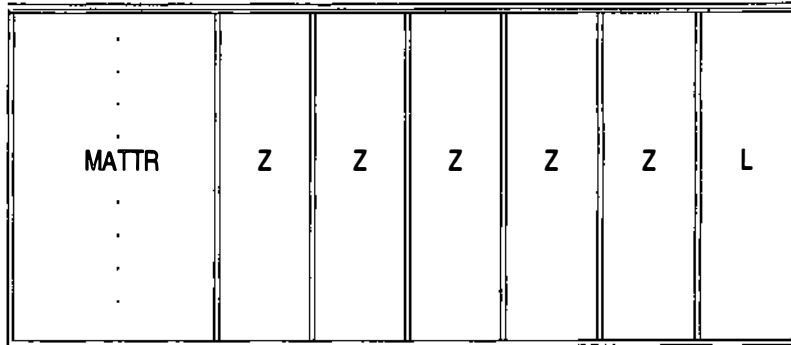


Figure 5-28 Mod type

MATTR is a 2-bit subfield that specifies the following module attributes:

MATTR	module attribute
0	Non-main module with no start addr
1	Non-main module with start addr
2	Main module with no start addr
3	Main module with start addr

“L” indicates whether the start addr field is interpreted as a logical address that requires fixing up by ld (L=1). Note that with ld, L must always equal 1.

“Z” indicates that this bit has not currently been assigned a function. These bits are required to be zero.

Physical start addresses (L=0) are not supported.

The start addrs field (present only if MATTR is 1 or 3) has the following format:

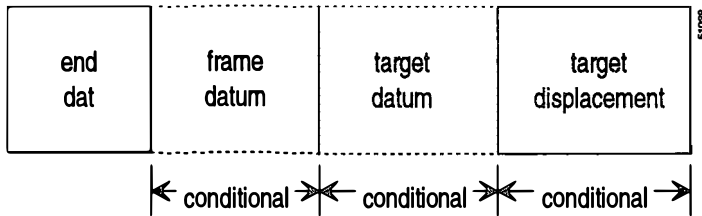


Figure 5-29 Start addrs

The starting address of a module has all the attributes of any other logical reference found in a module. The mapping of a logical starting address to a physical starting address is done in exactly the same manner as mapping any other logical address to a physical address as specified in the discussion of fixups and the FIXUPP record. The above subfields of the start addrs field have the same semantics as the FIX DAT, frame datum, target datum, and target displacement fields in the FIXUPP record. Only “primary” fixups are allowed. Frame method F4 is not allowed.

The target displacement field is 2 bytes in a 16-bit MODEND record (Rectyp=8AH) and 4 bytes in a 32-bit MODEND record (Rectyp=8BH).

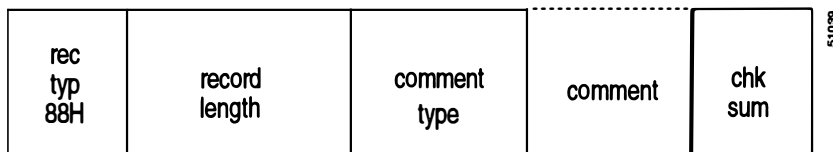


Figure 5-30 Comment record (COMMENT)

This record allows translators to include comments in object text.

Comment type

This field indicates the type of comment carried by this record. This allows comments to be structured for those processes that wish to selectively act on comments. The format of this field is as follows:

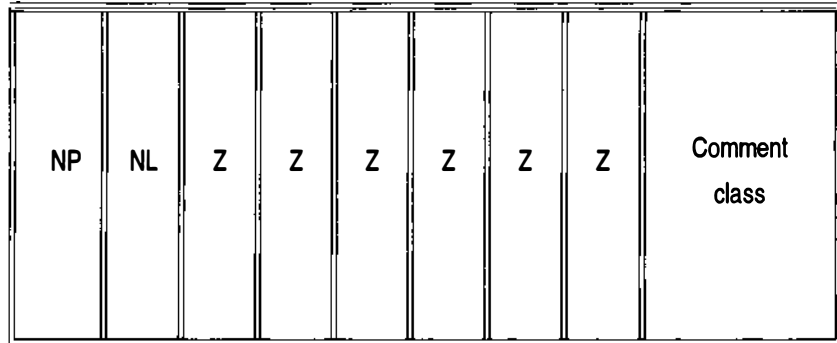


Figure 5-31 Comment type

The NP (nopurge) bit, if 1, indicates that it is not able to be purged by object file utility programs which implement the capability of deleting comment record.

The NL (nolist) bit, if 1, indicates that the text in the comment field is not to be listed in the listing file of object file utility programs which implement the capability of listing object comment records. In the above diagram, "Z" indicates no value and must equal zero.

The comment class field is defined as follows:

- 0 Language translator comment.
- 1 Intel copyright comment. The NP bit must be set.
- 2-155 Reserved for Intel use. (See Note 1 below.)
- 156-255 Reserved for users. Intel products will apply no semantics to these values. (See Note 2 below.)

NOTE

1. Class value 159 is used to specify a library to add to the link editor's library search list. The comment field will contain the name of the library. Note that unlike all other name specifications, the library name is not prefixed with its length. Its length is determined by the record length.
2. Class value 156 is used to specify a DOS level number. When the class value is 156, the comment field will contain a two-byte integer specifying a DOS level number.
3. Class value 161 is used to indicate that the module contains UNIX System V extensions to OMF, such as the various 32-bit record types.

Type representations for communal variables

This section defines the UNIX System V standard for communal variable allocation on the 8086 and 80286.

A communal variable is an uninitialized public variable whose final size and location are not fixed at compile time. Communal variables are similar to FORTRAN common blocks in that if a communal variable is declared in more than one object module being linked together, then its actual size will be the largest size specified in the several declarations. In the C language, all uninitialized public variables are communal. The following example shows three different declarations of the same C communal variable:

Example 5-1 C communal variable

```
char    foo[4];           /* In file a.c */
char    foo[1];          /* In file b.c */
char    foo[1024];       /* In file c.c */
```

If the objects produced from a.c, b.c, and c.c are linked together, then the linker will allocate 1024 bytes for the char array "foo."

A communal variable is defined in the object text by an external definition record (EXTDEF) and the type definition record (TYPDEF) to which it refers.

The TYPDEF for a communal variable has the following format:

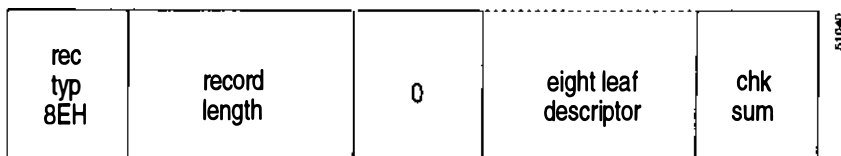


Figure 5-32 Typdef

The eight-leaf descriptor field has the following format:

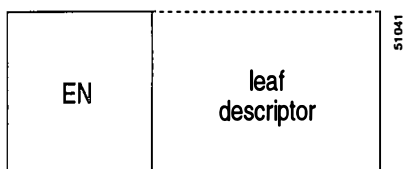


Figure 5-33 Eight-leaf descriptor field

The EN field specifies whether the next 8 leaves in the leaf descriptor field are EASY (bit = 0) or NICE (bit = 1). This byte is always zero for TYPDEFS for communal variables.

The leaf descriptor field has one of the following two formats. The format for communal variables in the default data segment (near variables) is as follows:

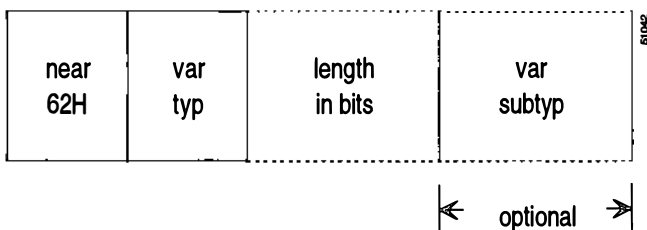


Figure 5-34 Leaf descriptor field

The VAR TYP>field may be either SCALAR (7BH), STRUCT (79H), or ARRAY (77H). The VAR SUBTYP field (if any) is ignored by ld. The format for communal variables not in the default data segment (far variables) is as follows:

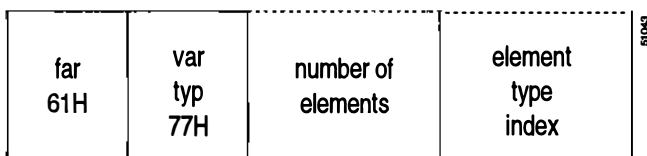


Figure 5-35 Variable type field

The variable type field must be array (77H). The length field specifies the number of elements, and the element type index is an index to a previously defined TYPDEF whose format is that of a near communal variable.

The format for the length in bits or number of elements fields is the same as the format for the leaf descriptor field, described in the TYPDEF record format section of this chapter.

Link time semantics

All EXTDEFs referencing a TYPDEF of the previously described formats are treated as communal variables. All others are treated as externally defined symbols for which a matching public symbol definition (<PUBDEF) is expected. A PUBDEF matching a communal variable definition will override the communal variable definition. Two communal variable definitions are said to match if the names given in the definitions match. If two matching definitions disagree as to whether a communal variable is near or far, the linker will assume the variable is near.

If the variable is near, then its size is the largest specified for it. If the variable is far, then the link editor issues a warning if there are conflicting array element size specifications; if there are no such conflicts, then the variable's size is the element size times the largest number of elements specified. The sum of the sizes of all near variables must not exceed 64K bytes. The sum of the sizes of all far variables must not exceed the size of the machine's addressable memory space.

"Huge" communal variables

A far communal variable whose size is larger than 64K bytes will reside in segments that are contiguous (8086) or have consecutive selectors (80286). No other data items will reside in the segments occupied by a huge communal variable.

If the linker finds matching huge and near communal variable definitions, it issues a warning message, since it is impossible for a near variable to be larger than 64K bytes.

The segmented x.out format

This section describes the executable object file format used in XENIX System V and in UNIX System V when the `-xenix` option is used. The format used is an extension to the existing *x.out* format, specifically enhanced for the segmented architecture of the 286 CPU. Note that *x.out* is a name for the format of the file, the actual executable file will be called *a.out* by default.

The UNIX System V linker (`/bin/ld`) links the Intel 86 relocatable object module format into the executable format described in this section.

The UNIX System V product supports a subset of segmented OMF. Other parts are specified here for use by other vendors, and to reserve their meaning for possible future use. Those parts supported in this release of UNIX System V are:

- The *x.out* header
- The *x.out* extended header
- The file segment table
- Multiple non-iterated text segments
- Multiple non-iterated data segments
- Symbol table segments in the format described herein

Note specifically that the machine-dependent table is *not* supported. The iterated text/data feature is supported by the kernel, but the UNIX System V linker will expand iterated records.

General description of x.out

The following is a general description of the *x.out* object file format, extended to handle segmentation. It implements iterated text and data segments, huge, large, middle, and small model, as well as block alignment to improve the efficiency of loading executable files.

NOTE The default file name produced by the linker is *a.out* regardless of the actual file format used. Any mention of *x.out* in this guide is referring only to the *format* of OMF executable files.

The extensions to the existing format consist of adding a file segment table that describes and points to various (possibly block-aligned) file segments. A file segment may contain a memory image, may indicate how to construct a memory image (iterated text or data), or may contain symbols or other non-executable information. In addition to the file segment table, there is an optional machine-dependent table.

The header must be first item in the object file, and the extended header must immediately follow the header. The extended header indicates the segment and (optional) machine-dependent tables' sizes and positions. Although the segment table is not block aligned, individual entries will line up on a multiple of 32 bytes (the size of a segment table entry). The segment table indicates the sizes and positions of the remaining file segments. The file segments may be aligned on a boundary that is a multiple of 512 bytes, with that multiple stored in the extended header, or at location zero if the file segments are not block aligned.

The segment table is an array of records describing the file segments, each containing:

- A segment type: text, data, symbols, and so on.
- Segment attributes, specific to the type of segment.
- A file pointer to the (possibly iterated) text/data for this segment.
- A physical size, the size of the segment in the file.
- A virtual size, the size the segment will occupy in memory.
- A location counter, this segment's current base address, usually 0.

A sample of a segment table entry is shown below. The *xs* fields in this data structure are referred to throughout the remaining discussion in this section.

Example 5-2 Segment table entry

```

struct xseg {
    unsigned short  xs_type;           /* x.out segment table entry */
    unsigned short  xs_attr;           /* segment type */
    unsigned short  xs_seg;            /* segment attributes */
    unsigned short  xs_seg;            /* segment number */
    char            xs_align;          /* log base 2 of alignment */
    char            xs_cres;           /* unused */
    long            xs_filpos;         /* file position */
    long            xs_psize;          /* physical size (in file) */
    long            xs_vsize;          /* virtual size (in core) */
    long            xs_rbase;          /* relocation base address */
    unsigned short  xs_noff;           /* segment name string table offset */
    unsigned short  xs_sres;           /* unused */
    long            xs_lres;           /* unused */
};

```

The segment table is a contiguous array of the above structures. Each file segment has a corresponding segment table entry that describes the segment's position *xs_filpos* and physical size *xs_psize* in the file. If there is no associated file segment, both fields must be set to zero.

The kernel's local descriptor table (LDT) can be built from the virtual size, the segment type, and segment attribute fields.

Example of file layout

This section provides an example of the layout of an *x.out* file where:

- The segment table has two entries (segments).
- The file page size is 512 bytes.

- Both file segments are smaller than 512 bytes.
- The second file segment contains iterated data.

Accessing the machine-dependent table and the file segment table must always be done through the absolute file pointers in the extended header. The ordering of the two tables and file segments shown above is not required to be consistent with the *x.out* UNIX System V specification.

Iterated segments

The data structure for an iterated segment is shown below:

Example 5-3 Iterated segment

```
struct xiter {                /* x.out iteration record */
    long    xi_size;          /* source byte count */
    long    xi_rep;           /* replication count */
    long    xi_offset;        /* destination offset in segment */
};
```

If the segment contains iterated text/data (indicated by a bit in the *xs_attr* field), the *xs_filpos* field is the file position of some number of iteration records mixed with the text/data to be iterated. If any part of a segment is iterated, then all of that segment is represented as iterated; non-iterated portions may be represented by an iteration record with a replication count of one.

The format of the text/data to be iterated is:

<iteration record> <text/data> <iteration record> <text/data> ...

where each **<iteration record>** is of the above **struct xiter** data structure. Each iteration record is followed by *xi_size* bytes of text/data that are to be placed in the current segment at the specified offset *xi_offset* *xi_rep* times. When *xs_psize* bytes of iteration records and text/data have been expanded, the iteration is complete.

Under UNIX System V, areas of memory that are initialized by more than one iteration record will have the contents of those memory areas undefined. Areas of memory that are not initialized by any iteration records will be zeroed out. An iteration byte count *xi_size* of zero will not result in any iteration. Portions of a segment that are to be **bss** sections should use an iteration record with a non-zero byte count and replicate one or more zeroed data bytes. For more information on **bss** sections, see the chapter on the common object file format in this book.

This representation of iterated text/data will handle iterations that contain very large replication counts and/or very large non-iterated sizes.

Non-iterated segments and implicit bss

If the iteration bit in `xs_attr` is *not* set, no iterations are required to initialize the segment. If the implicit bss bit in the `xs_attr` field is set and the virtual size is greater than the physical size, then the rest of the segment (up to `xs_vsize` bytes) is filled with zeros by the kernel loader. This implicit bss definition means that small and middle model executables' single data segments may still contain unexpanded bss without the use of explicit iteration records.

Segments made up entirely of implicit "C" bss need only set the physical size to zero, and set the implicit bss bit.

This eliminates the need for any file segment containing data or iteration records. If there are no iterations and no implicit bss, the virtual size of the segment `xs_vsize` must be the same as the physical size `xs_psize`, and a single copy of the text/data located at `xs_filpos` is all that is required to initialize the segment.

Large model

With *x.out* format, large model is supported by allowing multiple logical text and/or data segments. Middle and small models are simpler cases, with perhaps single logical segments for data (or both text and data). Iterated segments are independent of memory model.

Special header fields

The model bits in the `x_renv` field of the main header, `XE_LDATA` and `XE_LTEXT`, usually indicate the default size of data and text pointers used in the executable code. The kernel depends on these two bits to indicate the size of data and text pointers passed in system calls. However, since multiple segments are allowed in small and middle model, there can be little other meaning attached to these bits. Passing near data and/or text pointers implies use of the first data and text segments, respectively.

Also in the `x_renv` field, the absolute bit, `XE_ABS`, identifies a standalone executable file. When this bit is set, the extended header stack size field is used as the default physical load address. The UNIX System V kernel loader will not load a binary if the `XE_ABS` bit is set. The UNIX System V boot loader will not load a binary unless the `XE_ABS` bit is set. See the manual page on `ld(CP)` for information about how to set the `XE_ABS` bit and the physical load address.

Symbol table

The data structure for the *x.out* symbol table described in */usr/include/sys/relsym.h* is shown below:

```

struct sym {
    unsigned short s_type;    /* symbol management */
    unsigned short s_seg;    /* type */
    unsigned short s_seg;    /* file segment table index */
    long          s_value;    /* address or value */
};

```

The symbol table differs from the previous *x.out* format only in that the *s_seg* field now holds the selector of the segment that defines the symbol. If the symbol is absolute, the value field holds the symbol's value; otherwise, it holds the offset in the indicated segment to which the symbol refers.

The symbol name trails the above `struct sym` data structure in the form of a null terminated string. The type field values are defined in */usr/include/sys/relsym.h*.

The use of the *xs_seg* field in the segment table is undefined for symbol table segments. Its use may be defined by the particular symbol table format used.

UNIX System V executable format

UNIX System V does not execute binaries that make use of selectors below 0x3f or selectors that do not have the low 3 bits set (LDT, ring 3). UNIX System V also requires that the first data selector be after the last text selector. Binaries are allowed to have zero length segments or "holes" (unused selectors) in text or data, but holes in text may not contain data selectors, and holes in data may not contain text selectors.

The fields, *xext.xe_eseg*:*xexec.x_entry*, must contain the initial *cs:ip* addresses of the user process. *cs:ip* are the addresses of the starting segments of the program to be run.

Small-model impure binaries (text and data combined into a single segment) must have a single file segment, of type data, with a selector of at least 0x47. It must contain all text, followed by all data, followed by bss. The sizes of each must be stored in the *x_text*, *x_data*, and *x_bss* fields of the main header. UNIX System V uses the value stored in the *xext.xe_eseg* field as the text selector, which must be at least 0x3f and less than the data selector. All text/data/bss binaries are executable through the text selector, and all text/data/bss binaries are readable and writable through the data selector. UNIX System V maps the text selector to the same memory as the data selector.

In addition to the above, the linker, `ld`, generates binaries that conform to the following:

- Text selectors start at 0x3f.
- Data selectors start at the first free selector past text.
- All text selectors are contiguous.
- All data selectors are contiguous.
- Small-model impure binaries conform to the above specification, with 0x47 as the data selector. In the symbol table, the selector 0x47 is associated with data symbols, and the selector 0x3f is associated with text symbols, to allow `adb` and `nm` to present consistent data to the user.

Selected portions of include files

The following are selected portions of the `usr/include/sys/x.out.h` and `usr/include/sys/relsym.h` include files:

Example 5-4 x.out include files

```
struct xexec {
    unsigned short x_magic; /* x.out header */
    unsigned short x_ext; /* size of header extension */
    long x_text; /* size of text segment (s) */
    long x_data; /* size of initialized data (s) */
    long x_bss; /* size of uninitialized data (s) */
    long x_syms; /* size of symbol table (s) */
    long x_reloc; /* relocation table length (s) */
    long x_entry; /* entry point, machine dependent */
    char x_cpu; /* cpu type & byte/word order */
    char x_relsym; /* relocation & symbol format (u) */
    unsigned short x_renv; /* run-time environment */
};
```

```

struct text {
    /* x.out header extension */
    /* The following 4 fields are UNUSED currently */
    long    xe_trsize;    /* size of text relocation (s) */
    long    xe_drsize;    /* size of data relocation (s) */
    long    xe_tbase;     /* text relocation base (u) */
    long    xe_dbase;     /* data relocation base (u) */
    /* End of unused fields */
    long    xe_stksize;    /* stack size (if XE_FS set) */
    /* The following must be present if XE_SEG */
    long    xe_segpos;     /* segment table position */
    long    xe_segsize;    /* segment table size */
    long    xe_mdtpos;     /* machine dependent table position */
    long    xe_mdtsize;    /* machine dependent table size */
    char    xe_mdtttype;   /* machine dependent table type */
    char    xe_pagesize;   /* file pagesize, in multiples of 512 */
    char    xe_ostype;     /* operating system type */
    char    xe_osvers;     /* operating system version */
    unsigned short xe_eseg; /* entry segment, machine dependent */
    unsigned short xe_sres; /* reserved */
};

/*
 *      Definitions for xexec.x_renv (short).
 *
 *      vv      version compiled for
 *      xx      extra (zero)
 *      s      set if segmented x.out
 *      a      set if absolute (set up for physical address)
 *      i      set if segment table contains iterated text/data
 *      h      set if huge model data
 *      f      set if floating point hardware required
 *      t      set if large model text
 *      d      set if large model data
 *      o      set if text overlay
 *      f      set if fixed stack
 *      p      set if text pure
 *      s      set if separate I & D
 *      e      set if executable
 */

```

```

#define XE_V2      0x4000    /* up to and including 2.3 */
#define XE_V3      0x8000    /* after version 2.3 */
#define XE_VERS     0xc000    /* version mask */
#define XE_SEG      0x0800    /* segment table present */
#define XE_ABS      0x0400    /* absolute memory image (standalone) */
#define XE_ITER     0x0200    /* iterated text/data present */
#define XE_HDATA    0x0100    /* huge model data */
#define XE_VMOD     0x0100    /* virtual module */
#define XE_FPH      0x0080    /* floating point hardware required */
#define XE_LTEXT    0x0040    /* large model text */
#define XE_LDATA    0x0020    /* large model data */
#define XE_OVER     0x0010    /* text overlay */
#define XE_FS       0x0008    /* fixed stack */
#define XE_PURE     0x0004    /* pure text */
#define XE_SEP      0x0002    /* separate I & D */
#define XE_EXEC     0x0001    /* executable */

struct xseg {
    unsigned short xs_type;    /* x.out segment table entry */
    unsigned short xs_attr;    /* segment type */
    unsigned short xs_seg;     /* segment attributes */
    char          xs_align;    /* segment number */
    char          xs_cres;     /* log base 2 of alignment */
    long          xs_filpos;    /* unused */
    long          xs_psize;     /* file position */
    long          'xs_vsize;    /* physical size (in file) */
    long          xs_rbase;     /* virtual size (in core) */
    unsigned short xs_noff;     /* relocation base address/offset */
    unsigned short xs_sres;     /* segment name string table offset */
    long          xs_lres;     /* unused */
};

struct xiter {
    long          xi_size;      /* x.out iteration record */
    long          xi_rep;       /* source byte count */
    long          xi_offset;    /* replication count */
    long          xi_offset;    /* destination offset in segment */
};

```

```

struct sym (
    unsigned short s_type;      /* symbol management */
    unsigned short s_seg;      /* type */
    long s_value;              /* file segment table index */
                                /* address or value */
);
/*
 *      Definitions for xe_mdttype
 */
#define XE_MDTNONE 0           /* no machine dependent table */
#define XE_MDT286 1           /* iAPX286 LDT */

/*
 *      Definitions for xe_ostype
 */
#define XE_OSNONE 0
#define XE_OSXENIX 1          /* XENIX System V */
#define XE_OSIX86 2           /* iRMK */
#define XE_OSCCPM 3           /* Concurrent CP/M */

/*
 *      Definitions for xe_osvers
 */
#define XE_OSXV2 0            /* XENIX V3.x */
#define XE_OSXV3 1            /* XENIX System V */
#define XE_OSXV5 2            /* XENIX V5.x */

/*
 *      Definitions for xs_type:
 *      Values from 64 to 127 are reserved.
 */
#define XS_TNULL 0            /* unused segment */
#define XS_TTEXT 1            /* text segment */
#define XS_TDATA 2            /* data segment */
#define XS_TSYMS 3            /* symbol table segment */
#define XS_TREL 4             /* relocation segment */
#define XS_TSESTR 5           /* segment table's string table segment */
#define XS_TGRPS 6            /* group definitions segment */

#define XS_TIDATA 64           /* iterated data */
#define XS_TTSS 65            /* tss */
#define XS_TLFLX 66           /* lodfix */
#define XS_TDNAME 67           /* descriptor names */
#define XS_TDTEXT 68          /* debug text segment */
#define XS_TIDBG XS_TDTEXT
#define XS_TDFIX 69            /* debug relocation */
#define XS_TOVTAB 70           /* overlay table */
#define XS_T71 71
#define XS_TSYSTR 72           /* symbol string table */

```

```

/* Definitions for xs_attr: */
/* The top bit is set if the file segment represents */
/* a memory image. The other 15 bits' definitions */
/* depend on the type of file segment. */

#define XS_AMEM    0x8000 /* memory image segment */
#define XS_AMASK 0x7fff /* type specific field mask */

/* Definitions for xs_attr, built by or'ing the following */
/* bit patterns: these values are valid for XS_TTEXT and */
/* XS_TDATA file segments only. */

#define XS_AITER 0x0001 /* contains iteration records */
#define XS_AHUGE 0x0002 /* contains huge element */
#define XS_ABSS 0x0004 /* contains implicit bss */
#define XS_APURE 0x0008 /* is read-only, may be shared */
#define XS_AEDOWN 0x0010 /* segment expands downward */
#define XS_APRIV 0x0020 /* segment may not be combined */
#define XS_A32BIT 0x0040 /* segment is 32 bits */

/* These values are valid for XS_TSYMS file segments only. */

#define XS_S5BELL 0 /* Bell 5.2 format */
#define XS_SXSEG 1 /* x.out segmented format */
#define XS_SISLAND 2 /* island debugger support */

```

When using the `xs_seg` field, note that if the `XS_AMEM` bit is set in the `xs_attr` field, the file segment represents a memory image, and the value placed in this field should be the segment number as used by the hardware to reference the segment. This is the actual value placed in the segment register. For the 286, it is simply an LDT selector (under UNIX System V, if the privilege level is not 3, the file will not be executed). Otherwise the segment is not a memory image, and the contents of the field are not defined. File segments other than memory images may define and use this field as needed.

There are two bits in the `exec.x_cpu` field that are used to indicate the current byte and word ordering of the non-character data fields of the header, extended header, segment table, and symbol table. These bits, `XC_BSWAP` and `XC_WSWAP`, do not indicate the byte and word ordering of the target CPU, `XC_CPU`.

The segment table is not block aligned. No individual segment table entry may straddle a block boundary.

Definition of terms

The following terms are fundamental to the 8086 relocation and linkage.

- | | |
|---------------|--|
| Canonic | Any location in the 8086 MAS is contained in exactly 4096 distinct frames, but one of these frames can be distinguished because it has a higher frame number. This distinguished frame is called the "canonic frame" of the location. The canonic frame of a given byte is the frame so chosen that the byte's offset from that frame lies in the range 0 to 15 (decimal). Thus, if FOO is a symbol defining a memory location, one may speak of the "canonic frame of FOO," or of "FOO's canonic frame." By extension, if S is any set of memory locations, then there exists a unique frame that has the lowest frame number in the set of canonic frames of the locations in S. This unique frame is called the canonic frame of the set S. Thus, we may speak of the canonic frame of an LSEG, or of a group of LSEGs. |
| Class name | <p>LSEGs may optionally be assigned class names at translation time. Classes define a partition on LSEGs: two LSEGs are in the same class if they have the same class name.</p> <p>The link editor applies the following semantics to class names. The class name code or any class name whose suffix is code implies that all segments of that class contain only code and may be considered read-only. Such segments may be overlaid if the user specifies the module containing the segment as part of an overlay.</p> |
| Complete name | The complete name of an LSEG consists of the segment name, class name, and overlay name. LSEGs from different modules are combined if their complete names are identical. |
| Fixup | A "fixup" is some modification to object code, requested by a translator, performed by ld, achieving address-binding. |

NOTE This definition of "fixup" accurately represents the viewpoint maintained by ld. Nevertheless, the link editor can be used to achieve modifications of object code (that is, "fixups") that do not conform to this definition. For example, the binding of code to either hardware floating-point or software floating-point subroutines is a modification to an operation code, where the operation code is treated as if it were an address. The previous definition of "fixup" is not intended to disallow or disparage object code modifications.

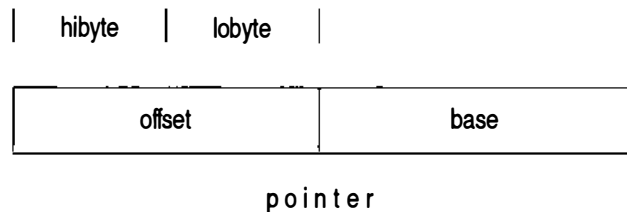
8086 translators specify a fixup with four data items:

- The place and type of a location to be fixed up.
- One of two possible fixup modes.
- A target, which is a memory address to which location must refer.
- A frame defining a context within which the reference takes place.

There are 5 types of location: a pointer, a base, an offset, a hbyte, and a lbyte.

The vertical alignment of the following figure illustrates four points. (Remember that the high-order byte of a word in 8086 memory is the byte with the higher address.) The `ld` command does not require the presence of the high- or low-order complement of these items. (For instance, in the case of hbyte, a high-order word, it doesn't matter if the low-order word is present.)

- A base is the high-order word of a pointer.
- An offset is the low-order word of a pointer.
- A hbyte is the high-order half of an offset.
- A lbyte is the low-order half of an offset.



Frame

A contiguous region of MAS that can be addressed using a single segment register. This concept is useful because the content of the four 8086 segment registers defines four (possibly overlapping) frames. No 16-bit address in the 8086 code can access a memory location outside of the current four frames. On an 8086, a frame must begin on a paragraph boundary (that is, a multiple of 16 bytes). On 80286 and 80386 processors, this restriction does not apply. On an 80386, a frame is a region of up to (2^{32}) bytes addressed by a single segment register.

Frame number	used only in reference to 8086 processors, or 80286/80386 processors operating in real address mode. Every frame begins on a paragraph boundary. The paragraphs in MAS can be numbered from 0 through 65535. These numbers, each of which defines a frame, are called frame numbers.
Group	A collection of LSEGs defined at translation time, whose final locations in MAS are constrained such that there is at least one frame that covers (contains) every LSEG in the collection. The notation Gr A(X,Y,Z) means that LSEGs X, Y, and Z form a group whose name is A. The fact that X, Y, and Z are all LSEGs in the same group does not imply any ordering of X, Y, and Z in MAS, nor does it imply any contiguity between X, Y, and Z. The link editor does not currently allow an LSEG to be a member of more than one group. The link editor ignores all attempts to place an LSEG in more than one group.
Indices	Index fields appear throughout this document. An index is an integer that selects some particular item from a collection of such items. Some examples are name index, segment index, group index, external index, and type index. In general, indices must assume values quite large (that is, much larger than 255). Nevertheless, a great number of object files will contain no indices with values greater than 50 or 100. Therefore, indices will be encoded in one or two bytes, as required. The high-order (left-most) bit of the first, and possibly the only, byte determines whether the index occupies one byte or two. If the bit is 0, then the index is a number between 0 and 127, occupying one byte. If the bit is 1, then the index is a number between 0 and 32K-1, occupying two bytes, and is determined as follows: the low-order 8 bits are in the second byte, and the high-order 7 bits are in the first byte.
Location	A location is specified by two data: (1) the location type, and (2) where the location is. The first is specified by the LOC subfield of the LOCAT field of the FIXUP record; the second is specified by the data record offset subfield of the LOCAT field of the FIXUP record. The link editor supports two fixup modes: "self-relative" and "segment-relative."

Self-relative fixups support the 8- and 16-bit offsets that are used in the CALL, JUMP and SHORT-JUMP instructions. Segment-Relative fixups support all other addressing modes of the 8086.

The target is the location in MAS being referenced. (More explicitly, the target may be considered the lowest byte in the object being referenced.) A target is specified in one of eight ways. There are four "primary" ways, and four "secondary" ways. Each primary way of specifying a target uses two kinds of data: an index-or-frame-number 'X', and a displacement 'D'.

- (T0) X is a segment index. The target is the Dth byte in the LSEG identified by the index.
- (T1) X is a group index. The target is the Dth byte in the LSEG identified by the index.
- (T2) X is an external index. The target is the Dth byte following the byte whose address is (eventually) given by the External Name identified by the index.
- (T3) X is a frame number. The target is the Dth byte in the frame identified by the frame number (that is, the address of target is $(X*16)+D$).

Each secondary way of specifying a target uses only one data item: the index-or-frame-number X. An implicit displacement equal to zero is assumed.

- (T4) X is a segment index. The target is the 0th (first) byte in the LSEG identified by the index.
- (T5) X is a group index. The target is the 0th (first) byte in the LSEG in the specified group that is eventually located lowest in MAS.
- (T6) X is an external index. The target is the byte whose address is the external name identified by the index.
- (T7) X is a frame number. The target is the byte whose 20-bit address is $(X*16)$.

NOTE The link editor does not support methods T3 and T7.

The following nomenclature is used to describe a target:

TARGET: SI (<segment name>), <displacement> [T0]

TARGET: GI (<group name>), <displacement> [T1]

TARGET: EI (<symbol name>), <displacement> [T2]

TARGET: SI (<segment name>) [T4]

TARGET: GI (<group name>) [T5]

TARGET: EI (<symbol name>) [T6]

The following examples illustrate how this notation is used:

TARGET: SI(CODE), 1024

The 1025th byte in the segment "CODE".

TARGET: GI(DATAAREA)

The location in MAS of a group called "DATAAREA".

TARGET: EI(SIN)

The address of the external subroutine "SIN".

TARGET: EI(PAYSCHEDULE), 24

The 24th byte following the location of an external data structure called "PAYSCHEDULE".

Every 8086 memory reference is to a location contained within some frame, where the frame is designated by the content of some segment register. For ld to form a correct, usable memory reference, it must know what the target is, and to which frame the reference is being made. Thus, every fixup specifies such a frame in one of six ways. Some use data X, which is in index-or-frame-number, as previously shown. Others require no data.

The six methods of specifying frames are:

1. (F0) X is a segment index. The frame is the canonic frame of the LSEG defined by the index.
2. (F1) X is a group index. The frame is the canonic frame defined by the group (that is, the canonic frame defined by the LSEG in the group that is eventually located lowest in MAS).
3. (F2) X is an external index. The frame is determined when the external name's public definition is found.

There are three cases:

- (F2a) The symbol is defined relative to some LSEG, and there is no associated group. The LSEG's canonic frame is specified.
 - (F2b) The symbol is defined absolutely, without reference to an LSEG, and there is no associated group. The frame is specified by the frame number subfield of the PUBDEF record that gives the symbol's definition.
 - (F2c) Regardless of how the symbol is defined, there is an associated group. The canonic frame of the group is specified. (The group is specified by the group index subfield of the PUBDEF Record.)
4. (F3) X is a frame number (specifying the obvious frame).
 5. (F4) No X. The frame is the canonic frame of the LSEG containing location.
 6. (F5) No X. The frame is determined by the target. There are four cases:
 - (F5a) The target specifies a segment index: in this case, the frame is determined as in (F0).
 - (F5b) The target specifies a group index: in this case, the frame is determined as in (F1).
 - (F5c) The target specifies an external index: in this case, the frame is determined as in (F2).
 - (F5d) The target is specified with an explicit frame number: in this case the frame is determined as in (F3).

NOTE The link editor does not support frame methods F2b, F3, or F5d.

Nomenclature describing frames is similar to the previous nomenclature for targets.

FRAME:	SI (<segment name>)	[F0]
FRAME:	GI (<group name>)	[F1]
FRAME:	EI (<symbol name>)	[F2]
FRAME:	LOCATION	[F4]
FRAME:	TARGET	[F5]
FRAME:	NONE	[F6]

For an 8086 memory reference, the frame specified by a self-relative reference is usually the canonic frame of the LSEG containing the location, and the frame specified by a segment relative reference is the canonic frame of the LSEG containing the target.

LSEG Logical Segment. A contiguous region of memory whose contents are determined at translation time (except for address-binding). Neither size nor location in MAS is necessarily determined at translation time. Size, although partially fixed, may not be final because the LSEG may be combined at link time with other LSEGs, forming a single LSEG.

On 8086/80286 processors, an LSEG must not be larger than 64K, so that it can fit in a frame. This means that any byte in an LSEG may be addressed by a 16-bit offset from the base of a frame covering the LSEG. An 80386 LSEG may be as much as (2^{32}) bytes in size and any byte in it may be addressed by a 32-bit offset from the base of the frame containing the LSEG.

MAS Memory Address Space.
Note that the MAS is distinguished from actual memory, which may occupy only a portion of the MAS.

Module An "inseparable" collection of object code and other information.

OMF Object Module Formats

Overlay name LSEGs may optionally be assigned overlay names. The overlay name of an LSEG is ignored by *ld* (version 2.40 and later versions), but it is used by Intel relocation and linkage products.

Paragraph number
equivalent to frame number.

PSEG Physical Segment. This term is equivalent to frame. Some people prefer PSEG to frame because the terms PSEG and LSEG reflect the physical and logical nature of the underlying segments.

PSEG number This term is equivalent to frame number.

Segment addressing

To have addressability of objects within a group, each group must be explicitly defined in the module. The group definition record (GRPDEF) provides a list of constituent segments either by segment name or by segment attribute such as "the segment defining symbol FOO" or "the segments with class name ROM."

The GRPDEF records within a module must follow all SEGDEF records because GRPDEF records can reference SEGDEF records when defining a group. The GRPDEF records must also precede all other records except header records, as the linker must process them first.

Segment name LSEGs are assigned segment names at translation time. These names serve two purposes:

- They play a role at link time in determining which LSEGs are combined with other LSEGs.
- They are used in assembly source code to specify groups.

Segment-relative fixup

Segment-relative fixups can be applied to any type of location.

The way in which a location containing a base component (that is, a base or a pointer) is fixed up depends on whether the code is to run in real or virtual address mode. The contents of the base portion of a location must ultimately be capable of being loaded into a segment register; therefore, in real address mode this will be a paragraph number and in virtual address mode this will be a selector value.

Fixup values for the base and offset components of a location are calculated as follows:

1. In real address mode:

The base fixup value (FBVAL) is defined as the frame number of the frame specified in the fixup.

The offset fixup value (FOVAL) is defined as the offset of the target from the start of the frame specified in the fixup. This offset must be ≥ 0 and $\leq \text{FFFF}$.

2. In protected mode:

The base fixup value (FBVAL) is defined as the segment selector of the frame specified in the fixup.

The offset fixup value (FOVAL) is defined as the offset of the target from the start of the frame specified in the fixup. This offset must be ≥ 0 and $\leq$ the maximum segment size implied by the segment selector for the frame; that is, $(2^{**16})-1$ for 80286 segments and 16-bit 80386 segments, or $(2^{**32})-1$ for 32-bit 80386 segments.

The fixup values for base and offset are applied to the location as follows:

1. If the location is a base or a pointer, then FBVAL is stored in the base component of the location.
2. If the location is a pointer, or a 16- or 32-bit offset, or a LOBYTE, then the offset fixup value (FOVAL) is added to the existing contents of the offset component of the location ignoring any overflow.
3. If the location is a HIBYTE, then FOVAL is divided by 256 and the result is added to the location, ignoring overflow.

Self-relative fixups

Self-relative fixups can be applied to locations which are either 16- or 32-bit offsets or they are LOBYTES. The result of applying a self-relative fixup to any other type of location is undefined.

Both the location and the target must lie within the frame specified for the fixup.

The value to be used in the fixup is defined as the displacement from the byte in memory following the location to the target.

If the location to be fixed-up is a LOBYTE, the fixup value must lie in the range -128 to 127.

If the location to be fixed up is a 16-bit offset, the fixup value must lie in the range -32768 to 32767.

The fixup value is added to the existing contents of the location, ignoring any overflow.

Self-relative fixups are typically applied to the relative displacement values used in instructions such as conditional jumps.

Symbol definition

The `ld` command supports three different types of records that fall into the class of symbol definition records. The two most important types are public names definition records (PUBDEFs) and external names definition records (EXTDEFs). These types are used to define globally visible procedures and data items and to resolve external references. In addition, TYPDEF records are used by `ld` for the allocation of communal variables. (See the section “Type Representations for Communal Variables” in this chapter.)

T-module A module created by a translator, such as C, Pascal, or FORTRAN.

The following restrictions apply to object modules:

- Every module needs a name. Translators provide names for T-modules, giving a default name (possibly the filename or a null name) if neither source code nor user specifies otherwise.
- Every T-module in a collection of linked modules must have a different name so that symbolic debugging systems can distinguish the various line numbers and local symbols. This restriction is not required by **ld**.

Chapter 6

C language compatibility with assembly language

This chapter explains how to use 8086/286/386 assembly-language routines with C-language programs and functions. In particular, it explains how to call assembly-language routines from C-language programs and how to call C-language functions from an assembly-language routine.

This assembly-language interface is especially useful for those assembly-language programmers who wish to use the functions of the standard C library and other C libraries.

NOTE Two different calling conventions are available. The 8086/80286 calling convention is established by configuring C-language programs with the **-M0**, **-M1**, or **-M2** option. The 80386 calling convention is established by configuring C-language programs with the **-M3** option.

C calling sequence for 8086/80286

To receive values from C-language function calls or to pass values to C functions, assembly-language routines must follow the C argument passing conventions. C-language function calls pass their arguments to the given functions by pushing the value of each argument onto the stack. The call pushes the value of the last argument first and the first argument last. If an argument is an expression, the call computes the expression's value before pushing it onto the stack.

Arguments with *char*, *int*, or *unsigned* type occupy a single word (16 bits) on the stack. Arguments with *long* type occupy a double word (32 bits) with the value's high-order word occupying the first word pushed onto the stack.

Arguments with *float* type are converted to *double* type (64 bits). Note that *char* type arguments are zero-extended to *int* type before being pushed on the stack.

If an argument is a structure, the function call pushes the last word of the structure first and each successive word in turn until the first word is pushed.

After a function returns control to a routine, the calling routine is responsible for removing arguments from the stack.

Entering an 8086/80286 assembly routine

Assembly-language routines that receive control from C function calls should preserve the contents of the **BP**, **SI**, and **DI** registers and set the **BP** register to the current **SP** register value before proceeding with their tasks. The following example illustrates the recommended instruction sequence for entry to an assembly-language routine:

Example 6-1 Entering an assembly-language routine (8086/80286)

```
entry:
    push    bp
    mov     bp, sp
    push    di
    push    si
```

This is the same sequence used by the C compiler.

If this sequence is used, the last argument passed by the function call (which is also the first argument given in the call's argument list) is at address "[bp+6]". Subsequent arguments begin at address "[bp+8]" or "[bp+10]" depending on the size of the first argument.

This sequence is strongly recommended even if the **SI** and **DI** registers are not modified, since it allows backtracing with the **adb** program during program debugging.

8086/80286 return values

Assembly-language routines that wish to return values to a C-language program or receive return values from C functions must follow the C return value conventions. C functions place return values that have *int*, *char*, or *unsigned* type in the AX register. They place values with *long* type in the AX and DX registers, with the high order word in DX.

To return a structure or a floating point value, C functions place the address of the given value in the AX register. The structure or floating point value must be in a static area in memory. Long addresses are returned in the AX and DX registers with the segment selector in DX.

Exiting an 8086/80286 routine

Assembly-language routines that return control to C programs should restore the values of the BP, SI, and DI registers before returning control. The following example illustrates the recommended instruction sequence for exiting a routine:

Example 6-2 Exiting an assembly-language routine (8086/80286)

```
pop    si
pop    di
mov    sp, bp
pop    bp
ret
```

This sequence does not change the AX, BX, CX, or DX registers or any of the segment registers. It also does not remove arguments from the stack. This is the responsibility of the calling routine.

8086/80286 program example

To illustrate the assembly-language interface, consider the following example of a C function:

Example 6-3 C function (8086/80286)

```
add(i,j)
int i,j;
{
    return(i+j);
}
```

If written as an assembly-language routine, this function must save the proper registers, retrieve the arguments from the stack, add the arguments, place the return value in the AX register, then restore registers and return control. The following is a example of how the routine can be written:

Example 6-4 Assembly-language routine (8086/80286)

```
_add:
    push    bp
    mov     bp, sp
    push    di
    push    si

    mov     ax, [bp+4]
    add     ax, [bp+6]

    pop     si
    pop     di
    mov     sp, bp
    pop     bp
    ret
```

If, on the other hand, the C function is to be called by an assembly-language routine, the routine must contain instructions that push the arguments on the stack in the proper order, call the function, and clear the stack. It may then use the return value in the AX register. The following is an example of the instructions that can do this:

Example 6-5 Calling a C function from an assembly-language routine (8086/80286)

```

push    <j value>
push    <i value>
call    _add
add     sp, *4

```

Note that the C compiler does not preserve ES over calls. Assembly-language routines need not preserve ES and should not assume that it will be preserved if they make calls to routines written in C.

80386 C-Language calling sequence

To receive values from 80386 C-language function calls, or to pass values to 80386 C-language functions, assembly-language routines must follow the 80386 C-language argument-passing conventions.

C-language function calls pass arguments to the function by pushing each argument onto the stack. The call pushes the last function argument first and the first function argument last onto the stack. If an argument is an expression, the call computes the expression's value before pushing it onto the stack.

Arguments with *char*, *int*, *unsigned*, *short*, or *long* type occupy a doubleword (32 bits or 4 bytes) on the stack. Arguments with *float* type are converted to *double* type (64 bits or 8 bytes). Note that *char*, *unsigned char*, *short*, and *unsigned short* type arguments are sign extended or zero extended, respectively, to *int* type before being pushed onto the stack.

If an argument is a structure, the function call pushes the last word of the structure first and each successive word in turn until the first word of the structure is pushed onto the stack.

After a function returns control to the calling routine, the calling routine is responsible for removing all function arguments from the stack.

Entering an 80386 assembly-language routine

Assembly-language routines that receive control from 80386 C function calls should preserve the contents of the EBP, ESI, EDI, and EBX registers. In addition, the routines should set the EBP register to the current ESP register value before proceeding with their tasks. The following example illustrates a recommended instruction sequence for entry to an assembly-language routine:

Example 6-6 Entering an assembly-language routine

```
entry:
    push    ebp
    mov     ebp, esp
    push    edi
    push    esi
    push    ebx
```

Note that this is the same routine that the compiler uses after pushing the function arguments onto the stack.

If this sequence is used, the last function argument pushed by the function call (which is also the first argument in the function's argument list) is at address "[ebp+8]". Subsequent arguments are at address "[ebp+12]" or "[ebp+16]", depending on the size of the argument pushed onto the stack at "8[ebp]".

80386 return values

Assembly-language routines that return values to an 80386 C-language program or receive return values from 80386 C-language functions must follow the 80386 C-language return-value conventions. C-language functions place return values that have *int*, *char*, *unsigned*, *short*, and *long* types in the EAX register.

Floating-point values are returned to the top of the ndp 80287 stack. The following example shows the recommended instruction sequence for passing floating-point values:

Example 6-7 Passing floating-point values

```
float func(),f;
f = func(f)
        fld     DWORD PTR f
        sub     esp,8
        fstp    QWORD PTR [esp]
        call    func           ; result in ST(0)
        add     esp,8
        fstp    DWORD PTR f
```

The following example shows the recommended instruction sequence for returning floating-point values:

Example 6-8 Returning floating-point values

```
float fvalue;
return (fvalue);
        fld     fvalue           ; result in ST(0)
        pop     edx
        pop     esi
        pop     edi
        leave
        ret
```

C-language structure returns are returned to a buffer whose address is passed as a hidden first parameter.

The following example shows the recommended instruction:

Example 6-9 C-language structure returns

```
struct shape
{
    int stuff, to, fill, it, with;
} in, out, them();
out = them(in);

        sub     esp,20
        mov     edi,esp
        lea     edi,in          ; structure copy input
        mov     ecx,5           ; struct onto stack
        rep     movsd
        lea     eax,out         ; pass address of
        push    eax             ; assignment as extra "hidden"
        call    them            ;parameter
        add     esp,24
```

The following example shows the recommended instruction sequence for returning C-language structure returns:

Example 6-10 Returning C-language structure returns

```
struct shape source;
return shape;
        mov     edi,[ebp+8]
        mov     esi,source
        mov     ecx,5
        rep     movsd
        pop     ebx
        pop     esi
        pop     edi
        leave
        ret
```

Exiting an 80386 routine

Before returning control from an assembly-language routine to an 80386 C-language program, restore the EBP, ESI, EDI, and EBX registers. The following example illustrates the recommended instruction sequence for exiting a routine:

Example 6-11 Exiting an assembly-language routine (80386)

```
pop    ebx
pop    esi
pop    edi
leave
ret
```

This sequence does not save the EAX, ECX, or EDX register. These registers are scratch registers for use by the compiler. If the routine modifies segment register ES, SS, or DS, the routine must preserve the modified segment registers. The sequence does not remove arguments from the stack. This is the responsibility of the calling routine.

80386 program example

The following example illustrates an 80386 C-language function that can be written as an assembly-language routine. The function takes two integer arguments and adds them together, returning the resultant value.

Example 6-12 C-language interface with assembly language

```
int add(i, j)
int i, j;
{
    return(i + j);
}
```

If written as an assembly-language routine, this function must save the proper registers, retrieve the arguments from the stack, add the arguments, place the return value in the EAX register, then restore the proper registers and return control to the calling routine. The following is an example of how the routine can be written:

Example 6-13 Assembly-language routine (80386)

```

_add:
    push    ebp
    mov     ebp, esp
    push    edi
    push    esi
    push    ebx

    mov     eax, [ebp+8]
    add     eax, [ebp+12]

    pop     ebx
    pop     esi
    pop     edi
    mov     esp, ebp
    pop     ebp
    ret

```

NOTE In the above assembly-language routine, it is not necessary to save the contents of the ESI, EDI, and EBX registers because the routine does not modify their contents. If the ESI, EDI, or EBX register was modified by the routine, its contents must be saved.

If the C-language function is to be called by an assembly-language routine, the routine must contain instructions that push the arguments onto the stack in the proper order, call the function, and clear the stack. It can then use the return value in the EAX register. The following is an example of the instructions that perform this task:

Example 6-14 Calling a C-function from an assembly-language routine (80386)

```

push    <j value>
push    <i value>
call    _add
add     esp, 8

```

Chapter 7

Writing portable programs

The standard definition of the C programming language leaves many details to be decided in specific implementations of the language. These unspecified features of the language detract from its portability and must be studied when attempting to write portable C code.

Most of the issues affecting C portability arise from differences either in target-machine hardware or in compilers. C was designed to compile efficient code for the target machine (initially a Digital Equipment Corporation PDP-11®), so many of the language features not precisely defined are those that reflect a particular machine's hardware characteristics.

This chapter highlights the various aspects of C that may not be portable across different machines and compilers. It also briefly discusses the portability of a C program in terms of its environment. The environment is determined by the system calls and library routines a program uses during execution, file path names it requires, and other items not guaranteed to be constant across different systems.

The C language has been implemented on many different computers with widely different hardware characteristics, from small eight-bit microprocessors to large mainframes. This chapter is concerned with the portability of C code in the MS-DOS, XENIX, and System V programming environments. This is a more restricted problem to consider, since all MS-DOS, System V, and XENIX operating systems to date run on hardware with the following basic characteristics:

- ASCII character set
- Eight-bit bytes
- Two-byte or four-byte integers
- Two's-complement arithmetic

These features are not formally defined for the language and may not be found in all implementations of C. However, the remainder of this chapter is devoted to those systems where these basic assumptions hold.

The C-language definition contains no specification of how input and output are performed. These specifications are left to system calls and library routines on individual systems. Within the System V and XENIX systems there are system calls and library routines that can be considered portable. This version of the C compiler includes system calls and library routines that can be considered portable across System V, XENIX, and MS-DOS systems. The run-time library for the System V C compiler for MS-DOS is composed primarily of System V and XENIX compatible routines. By restricting the use of XENIX and System V routines to those included in the MS-DOS library, the System V programmer can develop MS-DOS programs in the System V environment; C programs written on MS-DOS are easily portable to XENIX or System V.

Program portability

A program is “portable” if it can be compiled and run successfully on different machines without alteration. There are many ways to write portable programs. One way is to avoid using inherently nonportable language features. Another is to isolate any nonportable interactions with the environment, such as I/O to nonstandard devices. For example, programs should avoid hard-coded path names unless a path name is common to all systems.

Files required at compile time (such as include files) may also introduce non-portability if the path names used are not the same on all machines. In some cases, include files containing machine-specific definitions can be used to make the source code itself portable.

Machine hardware

Differences in the hardware of the various target machines and differences in the corresponding C compilers cause the greatest number of portability problems. This section lists problems commonly encountered.

Byte length

By definition, the *char* data type in C must be large enough to hold as positive integers all members of a machine’s character set. For the machines described in this chapter, the *char* size is an eight-bit byte.

Word length

The size of the basic data types for a given implementation are not formally defined in the C language. Therefore, they often follow the most natural size for the underlying machine. It is safe to assume that *short* is no longer than *long*. Beyond that, no assumptions are portable. For example, on some machines *short* is the same length as *int*, whereas on others *long* is the same length as *int*.

Two areas where different *int* sizes affect program portability are the following:

1. Array indexing. For very large arrays, a variable of type *int* may not be long enough to store the indices of the highest-numbered array elements.
2. Pointer subtraction. On some machines, an *int* variable may not be long enough to store the results of pointer subtraction. See the section on "Pointers" in this chapter for more information about this problem.

Programs that need to assume the size of a particular data type should avoid hard-coded constants where possible. Such information can usually be written in a fairly portable way. For example, the maximum positive integer (on a two's-complement machine) can be obtained with the following directive:

```
#define MAXPOS ((int)(((unsigned)-1) >> 1))
```

This is preferable to the following code:

```
#ifdef PDP11
#define MAXPOS 32767
#else
.
.
.
#endif
```

To find the number of bytes in an *int*, use *sizeof(int)* rather than 2, 4, or some other nonportable constant.

Storage alignment

The C language defines no particular layout for storage of data items relative to each other. The layout for storage of structure elements, or unions within the structure or union, is also left undefined by the language.

Some processors require that data types longer than one byte be aligned on even-byte address boundaries. Others, such as the 8086/8088, have no such hardware restriction. However, even with these machines, most compilers

generate code that aligns words, structures, arrays, and long words on even addresses or on even long-word addresses. Therefore, the following code sequence may give different results, depending on specific alignment requirements on different machines:

```

struct s_tag {
    char c;
    int i;
};
printf("%d\n", sizeof(struct s_tag));

```

This variation in data storage has two major implications: data accessed as nonprimitive data types are not portable, and code that makes assumptions about the layout on a particular machine is not portable.

Therefore, unions containing structures are nonportable if the union is used to access the same data in different ways. Unions are only likely to be portable if they are used exclusively to store different data in the same space at different times. For example, if the following union were used to obtain four bytes from a long word, the code would not be portable to some machines:

```

union {
    char c[4];
    long lw;
} u;

```

The *sizeof* operator should always be used when reading and writing structures, as follows:

```

struct s_tag st;
.
.
.
write(fd, &st, sizeof(st));

```

Using the *sizeof* operator ensures portability of the source code, but does not produce a portable data file. Portability of data is discussed in the “Portability of Data” section in this chapter.

Byte order in a word

The variation in byte order in a word affects the portability of data more than the portability of source code. However, any program that makes use of knowledge of the internal byte order in a word is not portable. For example,

on some XENIX or System V systems there is an include file, *misc.h*, that contains the following structure declaration:

```
/*
 * structure to access an
 * integer in bytes
 */
struct {
    char  lobyte;
    char  hobyte;
};
```

With certain less-restrictive compilers, this declaration could be used to access the high- and low-order bytes of an integer separately and in a completely nonportable way. The correct way to do this is to use mask and shift operations to extract the required byte, as shown in the following example:

```
#define LOBYTE(i) (i & 0xff)
#define HIBYTE(i) ((i >> 8) & 0xff)
```

These definitions provide a portable way to extract the least-significant and the next-least-significant bytes of an integer. Since the *int* type can be either two or four bytes, depending on the machine, even these definitions do not provide a completely portable way to access the bytes of an *int*.

One result of the byte-ordering problem is that the following code sequence will not always perform as intended:

```
int c = 0;

read(fd, &c, 1);
```

On machines where the low-order byte is stored first, the value of *c* is the byte value read. On other machines, the byte is read into some byte other than the low-order one, so the value of *c* is different.

Bit fields

Bit fields are not implemented in all C compilers. The C compiler implements bit fields and allows them to have any length up to the size of a *long*. However, in many implementations no bit field may be larger than an *int*, and no bit field can overlap an *int* boundary. If necessary, the compiler will leave gaps and move to the next *int* boundary. To ensure portability no individual field should exceed 16 bits.

The C language makes no guarantees about whether bit fields are assigned left to right or right to left. Therefore, although bit fields may be useful for storing flags and other small data items, their use in unions to dissect bits from other data is definitely nonportable.

Pointers

The C language is fairly generous in allowing manipulation of pointers, to the extent that most compilers do not generate warnings for nonportable pointer operations. A common nonportable use of pointers is the use of casts to assign one pointer to another pointer of a different data type. This practice usually makes some assumption about the internal byte ordering and layout of the data type, and is therefore nonportable. In the following code, the byte order in the array *c* is not portable:

```
char c[4];
long *lp;

lp = (long *)&c[0];
*lp = 0x12345678L;
```

Code like this is usually unnecessary or invalid. It is acceptable, however, when the `malloc` function is used to allocate space for variables that do not have *char* type. The routine is declared as type *char*, and the return value is cast to the type to be stored in the allocated memory. If this type is not *char*, then a compiler may issue a warning concerning illegal type conversion. In addition, the `malloc` function is designed always to return a starting address suitable for storing all types of data. A compiler may not know this, so it may give an additional warning about possible data-alignment problems. In the following example, `malloc` is used to obtain memory for an array of 50 integers:

```
extern void *malloc( );
int *ip;

ip = (int *)malloc(50 * sizeof(int));
```

This example will elicit a warning message from some compilers.

The *C Language Reference* states that a pointer can be assigned (or cast) to an integer large enough to hold it. Note that the size of the *int* type depends on the given machine and implementation. This type is *long* on some machines and *short* on others. The size may also be modified by *near* and *far* declarations. In general, do not assume that the following statement is always true:

```
sizeof(char *) == sizeof(int)
```

For example, the following construction is nonportable, assuming that the function identifier *func* is not previously declared:

```
int p;
p = (char *)func( );
```

This example assumes that a *char* pointer has the same length as an *int*.

Another consequence of different-sized *int* types on different machines is that pointer subtraction may not give the expected results. As an example of this case, subtracting pointers to the beginning and end of a very large array may give a result that is too large to store in an *int* variable, as shown in the following example:

```
int arr[20000], *b = arr, *e = &arr[20000];
int diff;
diff = e - b;    /* result too large to store in
                  int variable diff */
```

To correct this problem, coerce the result of the pointer subtraction *long* type, then assign the result to a variable of *unsigned int* type, as shown in the following example:

```
unsigned int udiff;
udiff = (long) ((int huge *)e - (int huge *)b);
```

In most implementations, the null pointer value *NULL* is defined to be the *int* value 0. The length of the 0 value can lead to problems for functions that expect pointer arguments longer than an *int*. For portable code, always use the following form to pass a *NULL* value of the correct size:

```
func( (char *)NULL );
```

Address space

The address space available to a program varies considerably from system to system. Some small processors allow only 64K for program text and data combined. Others allow up to 64K of data and 64K of program text. Larger machines may allow considerably more text and possibly more data as well.

Large programs, or programs that require large data areas, may have portability problems on small machines.

Character set

The C language does not require the use of the ASCII character set. In fact, the only character-set requirements are that all characters must fit in the *char* data type, and all characters must have positive values.

In the ASCII character set, all characters have values between 0 and 127 and therefore can be represented in seven bits. On an eight-bits-per-byte machine they are all positive, regardless of whether *char* is treated as signed or unsigned.

A set of character-classification macros is included as part of the run-time library for the C compiler. These macros should be used for most tests on character quantities. The macros are defined in the include file *ctype.h*, and are described in the *C Library Guide*.

The character-classification macros provide insulation from the internal structure of the character set. In addition, the names of the macros are often more meaningful than the equivalent line of code. Compare the following two lines:

```
if(isupper(c))

if((c >= 'A') && (c <= 'Z'))
```

With some of the other macros, such as *isxdigit* to test for a hexadecimal digit, the advantage is even greater. Also, the internal implementation of the macros makes them more efficient than an explicit test with an if statement.

Compiler differences

There are a number of C compilers running under various operating systems. The main areas of differences between compilers are outlined in this section.

Signed/unsigned char and sign extension

The current state of the signed versus unsigned *char* problem is best described as unsatisfactory. The sign-extension problem is a serious barrier to writing portable C, and the best solution at present is to write defensive code that does not rely on particular implementation features.

Shift operations

The left-shift operator (<<) shifts its operand a number of bits left, filling vacated bits with zeros. This is called a logical shift. When the right-shift operator (>>) is applied to an unsigned quantity, it performs a logical-shift operation; when it is applied to a signed quantity, the vacated bits may be filled with zeros (logical shift) or with sign bits (arithmetic shift). The decision is implementation dependent, and code that assumes a particular implementation is nonportable.

With compilers that use arithmetic right shift, it is necessary to shift and mask the appropriate number of high-order bits to avoid sign extension, as follows:

```
char c;

c = (c >> 3) & 0x1f;
```

You can also avoid sign extension by using the divide operator (/) as follows:

```
char c;

c = c / 8;
```

Identifier length

The use of long symbols and identifier names will cause portability problems with some compilers. To avoid these problems, a program should keep the following symbols as short as possible:

- C preprocessor symbols
- C local symbols
- C external symbols

Some loaders also place restrictions on the number of unique characters in C external symbols. Symbols unique in the first six characters are unique to most C-language processors.

In some C implementations, the case of letters in identifiers is not significant.

Register variables

The number and type of register variables in a function depend on the machine hardware and the compiler. Excess and invalid register declarations are treated as nonregister declarations and should not cause a portability problem. On an 8086, 8088 or 80286 processor, up to two register declarations are significant, and they must be applied to types of size *int* or smaller. On the 80386 processor, three register declarations are significant.

Since the compiler ignores excess variables of *register* type, the most important *register*-type variables should be declared first. In this way, register variables that the compiler ignores will be those that are the least important.

Type conversion

The C language has some rules for implicit type conversion; it also allows explicit type conversions by type casting. The most common portability problem in implicit type conversion is unexpected sign extension. This is a potential problem whenever something of type *char* is compared with an *int*.

The following example will never evaluate true on a machine that sign-extends *char* types but treats hexadecimal numbers as unsigned:

```
char c;

if(c == 0x80) {
    .
    .
    .
}
```

The following construction is also nonportable:

```
char c;
unsigned int u;

if (u == (unsigned)c) {
    .
    .
    .
}
```

Two problems can arise in the preceding example:

1. The *char* type may be considered either signed or unsigned, depending on the implementation.
2. For implementations that consider the *char* type to be signed, two different methods of carrying out the conversion are possible: the *char* value may be sign extended to *int* type first, then converted to *unsigned* type; or the *char* type may be converted to an unsigned type of the same size, then zero extended to *int* length.

The only safe comparison between *char* type and *int* is the following:

```
int c;

if(c == 'x') {
    .
    .
    .
}
```

This comparison is reliable because C guarantees all character constants to be positive.

Type conversion also occurs when arguments are passed to functions. Types *char* and *short* become *int*. Extending the *char* type can produce unexpected results. For example, the following program yields a result of -128 on some machines:

```
char c = 128;
printf("%d\n", c);
```

The unexpected negative value is produced because *c* is converted to *int* when it is passed to the `printf` function. The function itself has no knowledge of the original type of the argument and is expecting an *int*. The correct way to handle this situation is to code defensively and allow for the possibility of sign extension, as in the following example:

```
char c = 128;
printf("%d\n", c & 0xff);
```

Functions with a variable number of arguments

Functions with a variable number of arguments present a particular portability problem if the type of the arguments is also variable. In such cases the code is dependent on the size of various data types. For portability, these cases should be avoided.

Side effects and evaluation order

The C language makes few guarantees about the order of evaluation of operands in an expression or arguments to a function call. Therefore, the following statement is not portable:

```
func(i++, i++);
```

Even the following statement is unwise if `func` is ever likely to be replaced by a macro, since the macro may use *i* more than once:

```
func(i++);
```

Certain System V- and XENIX-compatible macros commonly appear in user programs; some of these use their argument only once, and therefore can safely be called with a side-effect argument. To determine whether a macro handles side effects correctly, examine the code for that macro to see whether or not the argument is evaluated more than once.

Operands to the following operators are guaranteed to be evaluated left to right:

`,` `&&` `||` `?:`

Note that the comma operator here is a separator for two C statements. A list of items separated by commas in a declaration list is not guaranteed to be processed left to right. Therefore, the following declaration on an 8086 or 8088 processor, where only two register variables may be declared, could give any two of the four variables *register* type, depending on the compiler:

```
register int a, b, c, d;
```

To give register storage to the most important variables, use separate declaration statements and declare the most important variables first. The order of processing of individual declaration statements is guaranteed to be sequential in the following statements:

```
register int a;
register int b;
register int c;
register int d;
```

Environment differences

Most programs make system calls and use library routines for various services. This section indicates some of those routines that are not always portable and those that particularly aid portability.

System calls specific to an operating system are not portable if they are not present on all other operating-system implementations of C. Most of the system calls defined in the UNIX System V run-time library are compatible with DOS system calls and are therefore portable to a DOS environment.

Any program is nonportable that contains hard-coded path names to files or directories, or that contains user identifier numbers, log-in names, terminal lines, or other system-dependent parameters. These types of constants should be in header files, passed as command-line arguments, or obtained from the environment.

Note that the members of the `printf` and `scanf` families of functions, including `fprintf`, `fscanf`, `printf`, `sprintf`, `scanf`, `vfprintf`, `vprintf`, `vsprintf`, and `sscanf`, have evolved in several ways, and some features are not completely portable. Some of the format-conversion characters have changed their meanings, in particular those relating to uppercase and lowercase in the output of hexadecimal numbers and the specification of *long* integers on 16-bit word machines. The C specifications for these routines are given in the *C Library Guide*.

Users should be wary of porting object files that reference the `setjmp` or `longjmp` functions from System V or XENIX to MS-DOS, unless these object files were compiled with the `-dos` option. The MS-DOS versions of these functions use a larger buffer size and may cause memory to be overwritten. Such object files can be ported from MS-DOS to System V without problems, and the corresponding source files can be ported in either direction.

Portability of data

Data files are almost always nonportable across different central-processing-unit (CPU) architectures. As mentioned above, structures, unions, and arrays have varying internal layout and padding requirements on different machines. In addition, byte ordering within words and actual word length may differ.

The only way to achieve data-file portability is to write and read data files as one-dimensional character arrays. This procedure prevents alignment and padding problems if the data are written and read as characters, and interpreted that way. Thus ASCII text files can usually be moved between different machine types without significant problems.

Type-size summary

Table 7.1 summarizes the sizes of the various data types as defined in the C compiler, Version 6.0.

Table 7-1 C type sizes

Type Name (Alternate Names)	Storage	Range of Values
char (signed char)	1 byte	-128 to 127
int (signed) (signed int)	Implementation dependent (2 bytes in UNIX C 5.1)	(-32,768 to 32,767 for UNIX C Version 5.1) (-2,147,483,648 to 2,147,483,647 for 386)
short (short int) (signed short) (signed short int)	2 bytes	-32,768 to 32,767
long (long int) (signed long) (signed long int)	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned ¹ char	1 byte	0 to 255
unsigned (unsigned int)	Implementation dependent (2 bytes in C 5.1)	(0 to 65,535 for C 5.1) (0 to 4,294,967,295 for 386)
unsigned short (unsigned short int)	2 bytes	0 to 65,535
unsigned long (unsigned long int)	4 bytes	0 to 4,294,967,295
enum	Implementation dependent (2 bytes in C 5.1)	(0 to 65,535 for C 5.1) (0 to 4,294,967,295 for 386)
float	4 bytes	Approximately 3.4E-38 to 3.4E+38 (7-digit precision)
double	8 bytes	Approximately 1.7E-308 to 1.7E+308 (15-digit precision)
long double	Implementation dependent (8 bytes in C 5.1)	Approximately 1.7E-308 to 1.7E+308 (15-digit precision)

1 Any type size modified by the *unsigned* keyword can be modified by the *signed* keyword instead. The *signed* keyword is useful if the -J option has been used to change the default sign of the *char* type.

Byte-ordering summary

Table 7.2 summarizes byte ordering for *short* and *long* types, respectively. The following conventions are used in these tables:

1. The lowest physically addressed byte of the data item is **a0**; **a1** has the byte address **a0 + 1**, and so on.
2. The least-significant byte of the data item is **b0**; **b1** is the next least significant, and so on.

Since byte ordering is machine specific, any program that actually makes use of the following information is guaranteed to be nonportable:

Table 7-2 Byte ordering comparison

CPU	Short type	Long type
8086	b0 b1	b0 b1 b2 b3
80286	b0 b1	b0 b1 b2 b3
PDP-11	b0 b1	b2 b3 b0 b1
VAX-11	b0 b1	b0 b1 b2 b3
M68000	b0 b1	b3 b2 b1 b0
Z8000	b1 b0	b3 b2 b1 b0

Chapter 8

Transport Layer Interface

This chapter provides detailed information on the UNIX System Transport Layer Interface and is intended for programmers who require the services defined by this interface. The “Examples” section near the end contains complete examples of programs discussed throughout the chapter. A working knowledge of UNIX System programming and data communication concepts is assumed. In particular, a working knowledge of the Reference Model of Open Systems Interconnection (OSI) is required. For an explanation of this model, see the *Encyclopedia* article “OSI Reference Model”.

The following acronyms are used throughout this chapter:

CLTS	Connectionless Transport Service
COTS	Connection Oriented Transport Service
ETSDU	Expedited Transport Service Data Unit
TSDU	Transport Service Data Unit

Overview of the transport layer interface

This section presents a high-level overview of the services of the Transport Layer Interface, which supports the transfer of data between two user processes. Figure 8-1 illustrates the Transport Layer Interface.

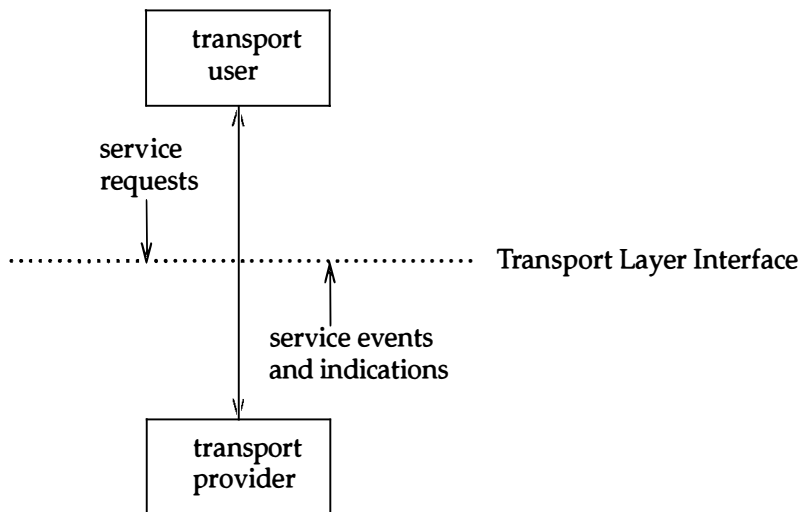


Figure 8-1 Transport layer interface

The transport provider is the entity that provides the services of the Transport Layer Interface, and the transport user is the entity that requires these services. An example of a transport provider is the ISO transport protocol, while a transport user may be a networking application or session layer protocol.

The transport user accesses the services of the transport provider by issuing the appropriate service requests. One example is a request to transfer data over a connection. Similarly, the transport provider notifies the user of various events, such as the arrival of data on a connection.

The Network Services Library of the UNIX System includes a set of functions that support the services of the Transport Layer Interface for user processes. These functions enable a user to initiate requests to the provider and process incoming events. Programs using the Transport Layer Interface can link the appropriate routines as follows:

```
cc prog.c -lnsl_s
```

Modes of service

Two modes of service, connection-mode and connectionless-mode, are provided by the Transport Layer Interface. Connection-mode is circuit-oriented and enables data to be transmitted over an established connection in a reliable, sequenced manner. It also provides an identification mechanism that avoids the overhead of address resolution and transmission during the data transfer phase. This service is attractive for applications that require relatively long-lived, datastream-oriented interactions.

Connectionless-mode, in contrast, is message-oriented and supports data transfer in self-contained units with no logical relationship required among multiple units. This service requires only a preexisting association between the peer users involved. This determines the characteristics of the data to be transmitted. All the information required to deliver a unit of data is presented to the transport provider, together with the data to be transmitted, in one service access (which need not relate to any other service access). Each unit of data transmitted is entirely self-contained. Connectionless-mode service is attractive for applications which:

- involve short-term request/response interactions
- exhibit a high level of redundancy
- are dynamically reconfigurable
- do not require guaranteed in-sequence delivery of data

Connection-mode service

The connection-mode transport service is characterized by four phases: local management, connection establishment, data transfer, and connection release.

Local management

The local management phase defines local operations between a transport user and a transport provider. For example, a user must establish a channel of communication with the transport provider, as illustrated in Figure 8-2. Each channel between a transport user and transport provider is a unique endpoint of communication, called the transport endpoint. The `t_open` routine enables a user to choose a particular transport provider that will supply the connection-mode services. This routine also establishes the transport endpoint.

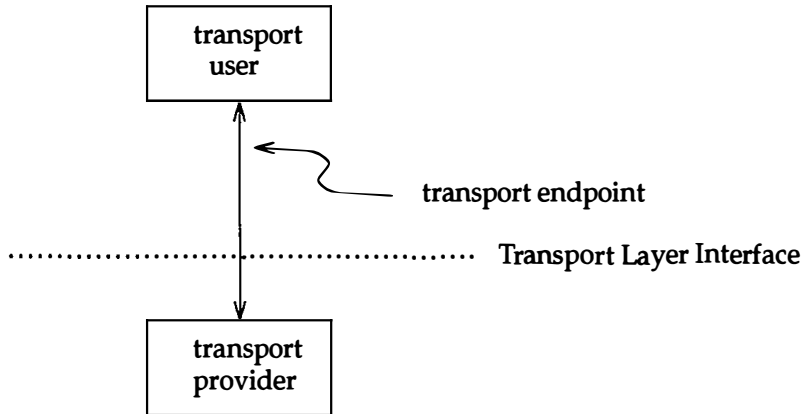


Figure 8-2 Channel between user and provider

Another necessary local function for each user is to establish an identity with the transport provider. Each user is identified by a transport address. More accurately, a transport address is associated with each transport endpoint, and one user process can manage several transport endpoints. In connection-mode service, one user requests a connection to another user by specifying that user's address. The structure of a transport address is defined by the address space of the transport provider. An address can be as simple as a random character string (for example, "file_server"), or as complex as an encoded bit pattern that specifies all information needed to route data through a network. Each transport provider defines its own mechanism for identifying users. Addresses can be assigned to each transport endpoint by `t_bind`.

In addition to `t_open` and `t_bind`, several routines are available to support local operations. Table 8-1 summarizes all local management routines of the Transport Layer Interface.

Table 8-1 Local management routines

Command	Description
t_alloc (S)	Allocates Transport Layer Interface data structures.
t_bind (S)	Binds a transport address to a transport endpoint.
t_close (S)	Closes a transport endpoint.
t_error (S)	Prints a Transport Layer Interface error message.
t_free (S)	Frees structures allocated using t_alloc .
t_getinfo (S)	Returns a set of parameters associated with a particular transport provider.
t_getstate (S)	Returns the state of a transport endpoint.
t_look (S)	Returns the current event on a transport endpoint.
t_open (S)	Establishes a transport endpoint connected to a chosen transport provider.
t_optmgmt (S)	Negotiates protocol-specific options with the transport provider.
t_sync (S)	Synchronizes a transport endpoint with the transport provider.
t_unbind (S)	Unbinds a transport address from a transport endpoint.

Connection establishment

The connection establishment phase enables two users to create a connection (virtual circuit) between them. This phase is illustrated by a client/server relationship between two transport users. One user, the server, typically advertises some service to a group of users and then listens for requests from those users. As each client requires the service, it attempts to connect itself to the server using the server's advertised transport address. The **t_connect** routine initiates the connect request. One argument to **t_connect**, the transport address, identifies the server the client wishes to access. The server is notified of each incoming request using **t_listen**, and can call **t_accept** to accept the client's request for access to the service. If the request is accepted, the transport connection is established.

Table 8-2 summarizes all routines available for establishing a transport connection.

Table 8-2 Connection establishment routines

Command	Description
<code>t_accept (S)</code>	Accepts a request for a transport connection.
<code>t_connect (S)</code>	Establishes a connection with the transport user at a specified destination.
<code>t_listen (S)</code>	Retrieves an indication of a connect request from another transport user.
<code>t_rcvconnect (S)</code>	Completes connection establishment if <code>t_connect</code> was called in asynchronous mode

Data transfer

The data transfer phase enables users to transfer data in both directions over an established connection. Two routines, `t_snd` and `t_rcv`, send and receive data over this connection. All data sent by a user is guaranteed to be delivered to the user on the other end of the connection in the order in which it was sent. Table 8-3 summarizes the connection-mode data transfer routines.

Table 8-3 Connection-mode data transfer routines

Command	Description
<code>t_rcv (S)</code>	Retrieves data that has arrived over a transport connection.
<code>t_snd (S)</code>	Sends data over an established transport connection.

Connection release

The connection release phase provides a mechanism for breaking an established connection. When you decide that the conversation should terminate, you can request that the provider release the transport connection. Two types of connection release are supported by the Transport Layer Interface. The first is an abortive release, which directs the transport provider to release the connection immediately. Any previously sent data that has not yet reached the other transport user may be discarded by the transport provider. The `t_snddis` routine initiates this abortive disconnect, and `t_rcvdis` processes the incoming indication of an abortive disconnect.

All transport providers must support the abortive release procedure. In addition, some transport providers may also support an orderly release facility that enables users to terminate communication gracefully with no data loss. The functions `t_sndrel` and `t_rcvrel` support this capability. Table 8-4 summarizes the connection release routines.

Table 8-4 Connection release routines

Command	Description
<code>t_rcvdis</code> (S)	Returns an indication of an aborted connection, including a reason code and user data.
<code>t_rcvrel</code> (S)	Returns an indication that the remote user has requested an orderly release of a connection.
<code>t_snddis</code> (S)	Aborts a connection or rejects a connect request.
<code>t_sndrel</code> (S)	Requests the orderly release of a connection.

Connectionless-mode service

The connectionless-mode transport service is characterized by two phases: local management and data transfer. The local management phase defines the same local operations described above for the connection-mode service.

The data transfer phase enables a user to transfer data units (sometimes called datagrams) to the specified peer user. Each data unit must be accompanied by the transport address of the destination user. Two routines, `t_sndudata` and `t_rcvudata`, support this message-based data transfer facility. Table 8-5 summarizes all routines associated with connectionless-mode data transfer.

Table 8-5 Connectionless-mode data transfer routines

Command	Description
<code>t_rcvudata</code> (S)	Retrieves a message sent by another transport user.
<code>t_rcvuderr</code> (S)	Retrieves error information associated with a previously sent message.
<code>t_sndudata</code> (S)	Sends a message to the specified destination user.

For additional information, see the following sections in the *Encyclopedia* article “Transport Layer Interface:”

- Transport layer interface states
- Outgoing events
- Incoming events
- Guidelines for protocol independence
- Transport user actions

State transitions

The Transport Layer Interface has two components:

- the library routines that provide the transport services to users
- the state transition rules that define the sequence in which the transport routines can be invoked

The state transition rules are presented in the form of state tables the the *Encyclopedia* article “Transport Layer Interface”. The state tables define the legal sequence of library calls based on state information and the handling of events. These events include user-generated library calls as well as provider-generated event indications.

NOTE Any user of the Transport Layer Interface must completely understand all possible state transitions before writing software using the interface.

Connection-mode service

This section describes the connection-mode service of the Transport Layer Interface. As discussed in the previous section, the connection-mode service can be illustrated using a client/server example. The important concepts of connection-mode service are presented using two programming examples. The first example illustrates how a client establishes a connection to a server and then communicates with the server. The second example shows the server’s side of the interaction. All examples discussed in this chapter are presented in their entirety in the section “Examples,” later in this chapter.

In the examples the client establishes a connection with a server process. The server then transfers a file to the client. The client receives the data from the server and writes it to its standard output file.

Local management

Before the client and server can establish a transport connection, each must first establish a local channel (the transport endpoint) to the transport provider using `t_open`, and establish its identity (or address) using `t_bind`.

The set of services supported by the Transport Layer Interface may not be implemented by all transport protocols. Each transport provider has a set of characteristics associated with it that determine the services it offers and the limitations associated with those services. This information is returned to the user by `t_open`, and consists of the following:

<code>addr</code>	maximum size of a transport address
<code>options</code>	maximum bytes of protocol-specific options that can be passed between the transport user and transport provider
<code>tsdu</code>	maximum message size that can be transmitted in either connection-mode or connectionless-mode
<code>etsdu</code>	maximum expedited data message size that can be sent over a transport connection
<code>connect</code>	maximum number of bytes of user data that can be passed between users during connection establishment
<code>discon</code>	maximum bytes of user data that can be passed between users during the abortive release of a connection
<code>servtype</code>	the type of service supported by the transport provider

The three service types defined by the Transport Layer Interface are:

<code>T_COTS</code>	The transport provider supports connection-mode service but does not provide the optional orderly release facility.
<code>T_COTS_ORD</code>	The transport provider supports connection-mode service with the optional orderly release facility.
<code>T_CLTS</code>	The transport provider supports connectionless-mode service.

Only one such service can be associated with the transport provider identified by `t_open`.

NOTE `t_open` returns the default provider characteristics associated with a transport endpoint. However, some characteristics can change after an endpoint has been opened. This occurs if the characteristics are associated with negotiated options (described later in this chapter). For example, if the support of expedited data transfer is a negotiated option, the value of this characteristic can change. `t_getinfo` can be called to retrieve the current characteristics of a transport endpoint.

Once a user establishes a transport endpoint with the chosen transport provider, it must establish its identity. As mentioned earlier, `t_bind` accomplishes this by binding a transport address to the transport endpoint. In addition, for servers, this routine informs the transport provider that the endpoint will be used to listen for incoming connect requests, also called connect indications.

An optional facility, `t_optmgmt(S)`, is also available during the local management phase. It enables a user to negotiate the values of protocol options with the transport provider. Each transport protocol is expected to define its own set of negotiable protocol options, which can include such information as Quality-of-Service parameters. Because of the protocol-specific nature of options, only applications written for a particular protocol environment are expected to use this facility.

The client

The local management requirements of the example client and server are used to discuss details of these facilities. The following are the definitions needed by the client program, followed by its necessary local management steps.

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

#define SRV_ADDR      1      /* server's well known address */

main()
{
    int fd;
    int nbytes;
    int flags = 0;
    char buf[1024];
    struct t_call *sndcall;
    extern int t_errno;

    if ((fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open failed");
        exit(1);
    }

    if (t_bind(fd, NULL, NULL) < 0) {
        t_error("t_bind failed");
        exit(2);
    }
}
```

The first argument to `t_open` is the pathname of a filesystem node that identifies the transport protocol that will supply the transport service. In this example, `/dev/tivc` is a STREAMS clone(S) device node that identifies a generic, connection-based transport protocol. The clone device finds an available minor device of the transport provider for the user. It is opened for both reading and writing, as specified by the `O_RDWR` open flag. The third argument can be used to return the service characteristics of the transport provider to the user. This information is useful when writing protocol-independent software (see the *Encyclopedia* article "Transport Layer Interface"). For simplicity, the client and server in this example ignore this information and assume that the transport provider has the following characteristics:

- The transport address is an integer value which uniquely identifies each user.
- The transport provider supports the `T_COTS_ORD` service type, and the example will use the orderly release facility to release the connection.

- User data cannot be passed between users during either connection establishment or abortive release.
- The transport provider does not support protocol-specific options.

Because these characteristics are not needed by the user, `NULL` is specified in the third argument to `t_open`. If the user needs a service other than `T_COTS_ORD`, another transport provider will be opened. An example of the `T_CLTS` service invocation is presented in the section “Connectionless-mode service,” later in this chapter.

The return value of `t_open` is an identifier for the transport endpoint that is used by all subsequent Transport Layer Interface function calls. This identifier is actually a file descriptor obtained by opening the transport protocol file (see `open(S)`).

After the transport endpoint is created, the client calls `t_bind` to assign an address to the endpoint. The first argument identifies the transport endpoint. The second argument describes the address the user would like to bind to the endpoint, and the third argument is set on return from `t_bind` to specify the address that the provider bound.

The address associated with a server’s transport endpoint is important because that is the address used by all clients to access the server. However, the typical client does not care what its own address is, because no other process will try to access it. That is the case in this example, where the second and third arguments to `t_bind` are set to `NULL`. A `NULL` second argument directs the transport provider to choose an address for the user. A `NULL` third argument indicates that the user does not care what address was assigned to the endpoint.

If either `t_open` or `t_bind` fails, the program calls `t_error` (see `t_error(S)`) to print an appropriate error message to `stderr`. If any Transport Layer Interface routine fails, the global integer `t_errno` is assigned an appropriate transport error value. A set of such error values is defined in `<tiuser.h>` for the Transport Layer Interface, and `t_error` prints an error message corresponding to the value in `t_errno`. This routine is analogous to `perror(S)`, which prints an error message based on the value of `errno`. If the error associated with a transport function is a system error, `t_errno` is set to `TSYSERR`, and `errno` is set to the appropriate value.

The server

The server in this example must take similar local management steps before communication can begin. The server must establish a transport endpoint through which it listens for connect indications. The necessary definitions and local management steps are shown below:

```
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>

#define DISCONNECT -1
#define SRV_ADDR 1 /* server's well-known address */

int conn_fd; /* connection established here */
extern int t_errno;

main()
{
    int listen_fd; /* listening transport endpoint */
    struct t_bind *bind;
    struct t_call *call;

    if ((listen_fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open failed for listen_fd");
        exit(1);
    } /* By assuming that the address is an integer value,
       this program may not run over another protocol.*/

    if ((bind = (struct t_bind *)t_alloc(listen_fd, T_BIND, T_ALL)) == NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }
    bind->qlen = 1;
    bind->addr.len = sizeof(int);
    *(int *)bind->addr.buf = SRV_ADDR;

    if (t_bind(listen_fd, bind, bind) < 0) {
        t_error("t_bind failed for listen_fd");
        _Exit(3);
    }
    /* Was the correct address bound?*/
    if (*(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\n");
        exit(4);
    }
}
```

As with the client, the first step is to call `t_open` to establish a transport endpoint with the desired transport provider. This endpoint, `listen_fd`, will be used to listen for connect indications. Next, the server must bind its well-known address to the endpoint. This address is used by each client to access the server. The second argument to `t_bind` requests that a particular address be bound to the transport endpoint. This argument points to a `t_bind` structure with the following format:

```
struct t_bind {
    struct netbuf addr;
    unsigned qlen;
}
```

where *addr* describes the address to be bound, and *qlen* indicates the maximum outstanding connect indications that can arrive at this endpoint. All Transport Layer Interface structure and constant definitions are found in `<tiuser.h>`.

The address is specified using a `netbuf` structure that contains the following members:

```
struct netbuf {
    unsigned int maxlen;
    unsigned int len;
    char *buf;
}
```

where *buf* points to a buffer containing the data, where *buf* points to a buffer containing the data, *len* specifies the bytes of data in the buffer, and *maxlen* indicates the maximum number of bytes that the buffer can hold. (This need be set only when data is returned to the user by a Transport Layer Interface routine.) For the `t_bind` structure, the data pointed to by *buf* identifies a transport address. It is expected that the structure of addresses will vary among each protocol implementation under the Transport Layer Interface. The `netbuf` structure is intended to support any such structure.

If the value of *qlen* is greater than 0, the transport endpoint can be used to listen for connect indications. In such cases, `t_bind` directs the transport provider to immediately begin queuing connect indications destined for the bound address. Furthermore, the value of *qlen* indicates the maximum outstanding connect indications the server wishes to process. The server must respond to each connect indication, either accepting or rejecting the request for connection. An outstanding connect indication is one to which the server has not yet responded. Often, a server will fully process a single connect

indication and respond to it before receiving the next indication. In this case, a value of 1 is appropriate for `qlen`. However, some servers may wish to retrieve several connect indications before responding to any of them. In such cases, `qlen` indicates the maximum number of such outstanding indications the server will process. An example of a server that manages multiple outstanding connect indications is presented in the section “Advanced topics”, later in this chapter.

`t_alloc` is called to allocate the `t_bind` structure needed by `t_bind`. `t_alloc` takes three arguments. The first is a file descriptor that references a transport endpoint. This is used to access the characteristics of the transport provider (see `t_open(S)`). The second argument identifies the appropriate Transport Layer Interface structure to be allocated. The third argument specifies which, if any, `netbuf` buffers should be allocated for that structure. `T_ALL` specifies that all `netbuf` buffers associated with the structure should be allocated; in this example, `T_ALL` causes the `addr` buffer to be allocated. The size of this buffer is determined from the transport provider characteristic that defines the maximum address size. The `maxlen` field of this `netbuf` structure is set to the size of the newly allocated buffer by `t_alloc`. The use of `t_alloc` helps to ensure the compatibility of user programs with future releases of the Transport Layer Interface.

The server in this example processes connect indications one at a time, and so `qlen` is set to one. The address information is then assigned to the newly allocated `t_bind` structure. This `t_bind` structure is used to pass information to `t_bind` in the second argument, and it is also used to return information to the user in the third argument.

On return, the `t_bind` structure contains the address that was bound to the transport endpoint. If the provider cannot bind the requested address (perhaps because it was bound to another transport endpoint), it chooses another appropriate address.

NOTE Each transport provider manages its address space differently. Some transport providers allow a single transport address to be bound to several transport endpoints, while others require a unique address per endpoint. The Transport Layer Interface supports either choice. Based on its address management rules, a provider determines if it can bind the requested address. If not, it chooses another valid address from its address space and binds it to the transport endpoint.

The server must check the bound address to ensure that it is the one previously advertised to clients. Otherwise, the clients will be unable to reach the server.

If `t_bind` succeeds, the provider begins queuing connect indications. This begins the next phase of communication, connection establishment.

Connection establishment

The connection establishment procedures highlight the distinction between clients and servers. The Transport Layer Interface imposes a different set of procedures in this phase for each type of transport user. The client initiates the connection establishment procedure by using `t_connect` to request a connection to a particular server. The server is then notified of the client's request by calling `t_listen`. The server can either accept or reject the client's request. It calls `t_accept` to establish the connection or calls `t_snddis` to reject the request. The client is notified of the server's decision when `t_connect` completes.

The Transport Layer Interface supports two facilities during connection establishment that may not be supported by all transport providers. The first is the ability to transfer data between the client and server when establishing the connection. The client can send data to the server when it requests a connection. This data is passed to the server by `t_listen`. Similarly, the server can send data to the client when it accepts or rejects the connection. The connect characteristic returned by `t_open` determines how much data, if any, two users can transfer during connection establishment.

The second optional service supported by the Transport Layer Interface during connection establishment is the negotiation of protocol options. The client can specify protocol options that it would like the remote user and/or transport provider to use. The Transport Layer Interface supports both local and remote option negotiation. As discussed earlier, option negotiation is inherently a protocol-specific function. Use of this facility is discouraged if protocol-independent software is a goal (see the section "Guidelines for protocol independence" in the *Encyclopedia* article "Transport Layer Interface").

The client

Continuing with the client/server example, the steps needed by the client to establish a connection are as follows:

```
/*
 * By assuming that the address is an integer value,
 * this program may not run over another protocol.
 */
if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL) {
    t_error("t_alloc failed");
    exit(3);
}
sndcall->addr.len = sizeof(int);
*(int *)sndcall->addr.buf = SRV_ADDR;

if (t_connect(fd, sndcall, NULL) < 0) {
    t_error("t_connect failed for fd");
    exit(4);
}
```

The **t_connect** call establishes the connection with the server. The first argument to **t_connect** identifies the transport endpoint through which the connection is established, and the second argument identifies the destination server. This argument is a pointer to a **t_call** structure, which has the following format:

```
struct t_call {
    struct netbuf addr;
    struct netbuf opt;
    struct netbuf udata;
    int sequence;
}
```

addr identifies the address of the server, **opt** can be used to specify protocol-specific options that the client would like to associate with the connection, and **udata** identifies user data that can be sent with the connect request to the server. The sequence field has no meaning for **t_connect**.

t_alloc is called above to allocate the **t_call** structure dynamically. Once allocated, the appropriate values are assigned. In this example, no options or user data are associated with the **t_connect** call, but the server's address must be set. The third argument to **t_alloc** is set to **T_ADDR** to indicate that an

appropriate `netbuf` buffer should be allocated for the address. The server's address is then assigned to `buf`, and `len` is set accordingly.

The third argument to `t_connect` can be used to return information about the newly established connection to the user, and it can retrieve any user data sent by the server in its response to the connect request. It is set to `NULL` by the client here to indicate that this information is not needed. The connection is established on successful return of `t_connect`. If the server rejects the connect request, `t_connect` fails and sets `t_errno` to `TLOOK`.

Event handling

The `TLOOK` error has special significance in the Transport Layer Interface. Some Transport Layer Interface routines may be interrupted by an unexpected asynchronous transport event on the given transport endpoint, and `TLOOK` notifies the user that an event has occurred. As such, `TLOOK` does not indicate an error with a Transport Layer Interface routine. Rather, it indicates that the normal processing of that routine will not be performed because of the pending event. The events defined by the Transport Layer Interface are listed here:

<code>T_LISTEN</code>	A request for a connection, called a connect indication, has arrived at the transport endpoint.
<code>T_CONNECT</code>	The confirmation of a previously sent connect request, called a connect confirmation, has arrived at the transport endpoint. The confirmation is generated when a server accepts a connect request.
<code>T_DATA</code>	User data has arrived at the transport endpoint.
<code>T_EXDATA</code>	Expedited user data has arrived at the transport endpoint. (Expedited data will be discussed later in this chapter.)
<code>T_DISCONNECT</code>	A notification that the connection was aborted or that the server rejected a connect request, called a disconnect indication, has arrived at the transport endpoint.
<code>T_ORDREL</code>	A request for the orderly release of a connection, called an orderly release indication, has arrived at the transport endpoint.
<code>T_UDERR</code>	The notification of an error in a previously sent datagram, called a unitdata error indication, has arrived at the transport endpoint (see the section "Connectionless-mode service," later in this chapter).

In some states it is possible to receive one of several asynchronous events. For a description of state transitions, see the section “Transport layer interface states” in the *Encyclopedia* article, “Transport Layer Interface”. The `t_look` routine (see `t_look(S)`) enables a user to determine what event has occurred if a `TLOOK` error is returned. The user can then process that event accordingly. In the example, if a connect request is rejected, the event passed to the client will be a disconnect indication. The client will exit if its request is rejected.

The server

Returning to the example, when the client calls `t_connect`, a connect indication is generated on the server’s listening transport endpoint. The steps required by the server to process the event are presented below. For each client, the server accepts the connect request and spawns a server process to manage the connection.

```
if ((call = (struct t_call *)t_alloc(listen_fd, T_CALL, T_ALL)) == NULL) {
    t_error("t_alloc of t_call structure failed");
    exit(5);
}

while (1) {
    if (t_listen(listen_fd, call) < 0) {
        t_error("t_listen failed for listen_fd");
        exit(6);
    }

    if ((conn_fd = accept_call(listen_fd, call)) != DISCONNECT)
        run_server(listen_fd);
}
}
```

The server loops forever, processing each connect indication. First the server calls `t_listen` to retrieve the next connect indication. When one arrives, the server calls `accept_call` to accept the connect request. `accept_call` accepts the connection on an alternate transport endpoint (as discussed below) and returns the value of that endpoint. `conn_fd` is a global variable that identifies the transport endpoint where the connection is established. Because the connection is accepted on an alternate endpoint, the server can continue listening for connect indications on the endpoint that was bound for listening. If the call is accepted without error, `run_server` spawns a process to manage the connection.

The server allocates a `t_call` structure to be used by `t_listen`. The third argument to `t_alloc`, `T_ALL`, specifies that all necessary buffers should be allocated for retrieving the caller’s address, options, and user data. As mentioned earlier, the transport provider in this example does not support the transfer of

user data during connection establishment, and also does not support any protocol options. Therefore, `t_alloc` does not allocate buffers for the user data and options. However, it must allocate a buffer large enough to store the address of the caller. `t_alloc` determines the buffer size from the *addr* characteristic returned by `t_open`. The `maxlen` field of each `netbuf` structure is set to the size of the newly allocated buffer by `t_alloc`. (`maxlen` is 0 for the user data and options buffers.)

Using the `t_call` structure, the server calls `t_listen` to retrieve the next connect indication. If one is currently available, it is returned to the server immediately. Otherwise, `t_listen` blocks until a connect indication arrives.

NOTE The Transport Layer Interface supports an asynchronous mode that prevents a process from blocking. This feature is discussed in the section “Advanced topics”, later in this chapter.

When a connect indication arrives, the server calls **accept_call** to accept the client's request, as follows:

```
accept_call(listen_fd, call)
int listen_fd;
struct t_call *call;
{
    int resfd;
    if ((resfd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open for responding fd failed");
        exit(7);
    }
    if (t_bind(resfd, NULL, NULL) < 0) {
        t_error("t_bind for responding fd failed");
        exit(8);
    }
    if (t_accept(listen_fd, resfd, call) < 0) {
        if (t_errno == TLOOK) { /* must be a disconnect */
            if (t_rcvdis(listen_fd, NULL) < 0) {
                t_error("t_rcvdis failed for listen_fd");
                exit(9);
            }
            if (t_close(resfd) < 0) {
                t_error("t_close failed for responding fd");
                exit(10);
            }
            /* go back up and listen for other calls */
            return(DISCONNECT);
        }
        t_error("t_accept failed");
        exit(11);
    }
    return(resfd);
}
```

accept_call takes two arguments. **listen_fd** identifies the transport endpoint where the connect indication arrived, and **call** is a pointer to a **t_call** structure that contains all information associated with the connect indication. The server first establishes another transport endpoint by opening the clone device node of the transport provider and binding an address. As with the client, a NULL value is passed to **t_bind** to specify that the user does not care what address is bound by the provider. The newly established transport endpoint, **resfd**, is used to accept the client's connect request.

The first two arguments of `t_accept` specify the listening transport endpoint and the endpoint where the connection will be accepted, respectively. A connection can be accepted on the listening endpoint. However, this prevents other clients from accessing the server for the duration of that connection.

The third argument of `t_accept` points to the `t_call` structure associated with the connect indication. This structure should contain the address of the calling user and the sequence number returned by `t_listen`. The value of *sequence* has particular significance if the server manages multiple outstanding connect indications. The section “Advanced topics” presents such an example. Also, the `t_call` structure should identify protocol options the user would like to specify and user data that can be passed to the client. Because the transport provider in this example does not support protocol options or the transfer of user data during connection establishment, the `t_call` structure returned by `t_listen` can be passed without change to `t_accept`.

For simplicity in the example, the server exits if either the `t_open` or `t_bind` call fails. `exit(S)` closes the transport endpoint associated with `listen_fd`, causing the transport provider to pass a disconnect indication to the client that requested the connection. This disconnect indication notifies the client that the connection was not established; `t_connect` then fails, setting `t_errno` to `TLOOK`.

`t_accept` can fail if an asynchronous event has occurred on the listening transport endpoint before the connection is accepted. In this case, `t_errno` is set to `TLOOK`. The state transition table in the *Encyclopedia* article “Transport Layer Interface”, shows that the only event that can occur in this state with only one outstanding connect indication is a disconnect indication. This event can occur if the client decides to undo the connect request it had previously initiated. If a disconnect indication arrives, the server must retrieve the disconnect indication using `t_rcvdis`. This routine takes a pointer to a `t_discon` structure as an argument which is used to retrieve information associated with a disconnect indication. However, in this example the server does not care to retrieve this information, and so it sets the argument to `NULL`. After receiving the disconnect indication, `accept_call` closes the responding transport endpoint and returns `DISCONNECT`, which informs the server that the connection was disconnected by the client. The server then listens for further connect indications.

Figure 8-3 illustrates how the server establishes connections:

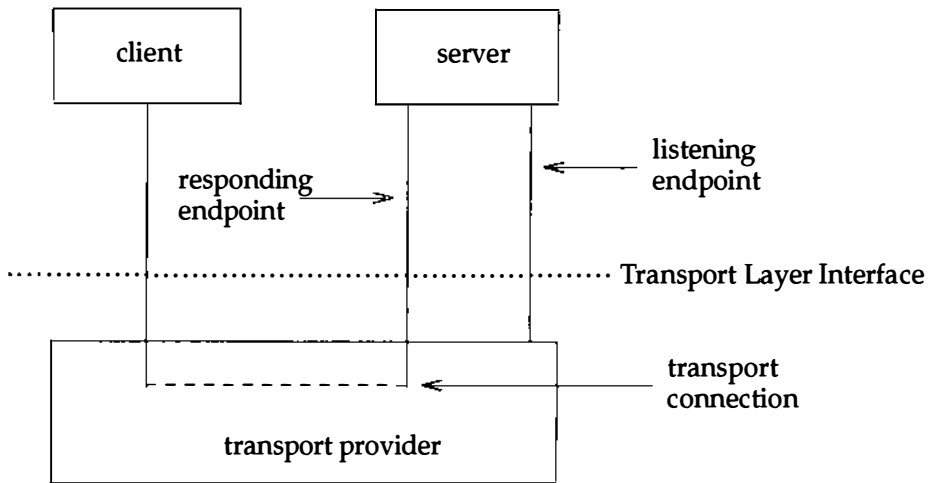


Figure 8-3 Listening and responding transport endpoints

The transport connection is established on the newly created responding endpoint, and the listening endpoint is freed to retrieve further connect indications.

Data transfer

Once the connection has been established, both the client and server can begin transferring data over the connection using `t_snd` and `t_rcv`. From this point on, the Transport Layer Interface does not differentiate the client from the server. Either user can send and receive data, or release the connection. The Transport Layer Interface guarantees reliable, sequenced delivery of data over an existing connection.

Two classes of data can be transferred over a transport connection: normal and expedited. Expedited data is typically associated with information of an urgent nature. The exact semantics of expedited data are subject to the interpretations of the transport provider. Furthermore, not all transport protocols support the notion of an expedited data class (see `t_open(S)`).

All transport protocols support the transfer of data in byte stream mode, where “byte stream” implies no concept of message boundaries on data which is transferred over a connection. However, some transport protocols support the preservation of message boundaries over a transport connection. This service is supported by the Transport Layer Interface, but protocol-independent software must not rely on its existence.

The message interface for data transfer is supported by a special flag of `t_snd` and `t_rcv` called `T_MORE`. The messages, called Transport Service Data Units (TSDU), can be transferred between two transport users as distinct units. The maximum size of a TSDU is a characteristic of the underlying transport protocol. This information is available to the user from `t_open` and `t_getinfo`. Because the maximum TSDU size can be large (possibly unlimited), the Transport Layer Interface enables a user to transmit a message in multiple units.

To send a message in multiple units over a transport connection, the user must set the `T_MORE` flag on every `t_snd` call except the last. This flag indicates that the user will send more data associated with the message in a subsequent call to `t_snd`. The last message unit should be transmitted with `T_MORE` turned off to indicate that this is the end of the TSDU.

Similarly, a TSDU can be passed to the user on the receiving side in multiple units. Again, if `t_rcv` returns with the `T_MORE` flag set, the user should continue calling `t_rcv` to retrieve the remainder of the message. The last unit in the message is indicated by a call to `t_rcv` that does not set `T_MORE`.

WARNING The `T_MORE` flag implies nothing about how the data is packaged below the Transport Layer Interface. Furthermore, it implies nothing about how the data is delivered to the remote user. Each transport protocol, and each implementation of that protocol, can package and deliver the data differently.

For example, if a user sends a complete message in a single call to `t_snd`, there is no guarantee that the transport provider will deliver the data in a single unit to the remote transport user. Similarly, a TSDU transmitted in two message units may be delivered in a single unit to the remote transport user. The message boundaries can be preserved only by noting the value of the `T_MORE` flag on `t_snd` and `t_rcv`. This guarantees that the receiving user will see a message with the same contents and message boundaries as was sent by the remote user.

The client

Continuing with the client/server example, the server now transfers a log file to the client over the transport connection. The client receives this data and writes it to its standard output file. The client and server use a byte stream interface, where message boundaries (that is, the `T_MORE` flag) are ignored. The client receives data using the following instructions:

```
while ((nbytes = t_rcv(fd, buf, 1024, &flags)) != -1) {
    if (fwrite(buf, 1, nbytes, stdout) < 0) {
        fprintf(stderr, "fwrite failed\n");
        exit(5);
    }
}
```

The client continuously calls `t_rcv` to process incoming data. If no data is currently available, `t_rcv` blocks until data arrives. `t_rcv` retrieves the available data up to 1024 bytes, which is the size of the client's input buffer, and returns the number of bytes that were received. The client then writes this data to standard output and continues. The data transfer phase completes when `t_rcv` fails. `t_rcv` fails if an orderly release indication or disconnect indication arrives, as discussed later in this chapter. If the `fwrite` call (see `fread(S)`) fails for any reason, the client exits, thereby closing the transport endpoint. If the transport endpoint is closed (either by `exit` or `t_close`) when it is in the data transfer phase, the connection is aborted and the remote user receives a disconnect indication.

The server

Looking now at the other side of the connection, the server manages its data transfer by spawning a child process to send the data to the client. The parent process then loops back to listen for further connect indications. The server calls `run_server` to spawn this child process as follows:

Example 8-1 Server connection (part 1/2)

```
connrelease()
{
    /* conn_fd is global because needed here */
    if (t_look(conn_fd) == T_DISCONNECT) {
        fprintf(stderr, "connection aborted\n");
        exit(12);
    }
    /* else orderly release indication - normal exit */
    exit(0);
}

run_server(listen_fd)
int listen_fd;
{
    int nbytes;
    FILE *logfp;          /* file pointer to log file */
    char buf[1024];

    switch (fork()) {
    case -1:
        perror("fork failed");
        exit(20);
    default:                /* parent */
        /* close conn_fd and then go up and listen again */
        if (t_close(conn_fd) < 0) {
            t_error("t_close failed for conn_fd");
            exit(21);
        }
        return;
    }
```

Example 8-2 Server connection (part 2/2)

```

case 0:          /* child */

    /* close listen_fd and do service */
    if (t_close(listen_fd) < 0) {
        t_error("t_close failed for listen_fd");
        exit(22);
    }
    if ((logfp = fopen("logfile", "r")) == NULL) {
        perror("cannot open logfile");
        exit(23);
    }

    signal(SIGPOLL, connrelease);
    if (ioctl(conn_fd, I_SETSIG, S_INPUT) < 0) {
        perror("ioctl I_SETSIG failed");
        exit(24);
    }
    if (t_look(conn_fd) != 0) {      /* was disconnect already there? */
        fprintf(stderr, "t_look returned unexpected event\n");
        exit(25);
    }

    while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
        if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
            t_error("t_snd failed");
            exit(26);
        }
}

```

After the **fork**, the parent process returns to the main processing loop and listens for further connect indications. Meanwhile, the child process manages the newly established transport connection. If the **fork** call fails, **exit** closes the transport endpoint associated with **listen_fd**. This action causes a disconnect indication to be passed to the client, and the client's **t_connect** call fails.

The server process reads 1024 bytes of the log file at a time and sends that data to the client using **t_snd**. **buf** points to the start of the data buffer, and **nbytes** specifies the number of bytes to be transmitted. The fourth argument is used to specify optional flags. Two flags are currently supported: **T_EXPEDITED** can be set to indicate that the data is expedited, and **T_MORE** can be set to define message boundaries when transmitting messages over a connection. Neither flag is set by the server in this example.

If the user begins to flood the transport provider with data, the provider can exert back pressure to provide flow control. In such cases, **t_snd** blocks until the flow control is relieved and then resumes its operation. **t_snd** will not complete until **nbyte** bytes have been passed to the transport provider.

The `t_snd` routine does not look for a disconnect indication before passing data to the provider. Also, because the data traffic is flowing in one direction, the user will never look for incoming events. If, for some reason, the connection is aborted, the user should be notified because data may be lost. One option available to the user is to use `t_look` to check for incoming events before each `t_snd` call. A more efficient solution is the one presented in the example. The STREAMS `I_SETSIG ioctl` enables a user to request a signal when a given event occurs (see `streamio(S)` and `signal(S)`). The STREAMS event of concern here is `S_INPUT`, which causes a signal to be sent to the user if any input arrives on the stream referenced by `conn_fd`. If a disconnect indication arrives, the signal catching routine (`connrelease`) prints an appropriate error message and then exits.

If the data traffic flowed in both directions in this example, the user would not have to monitor the connection for disconnects. If the client alternated `t_snd` and `t_rcv` calls, it could rely on `t_rcv` to recognize an incoming disconnect indication.

Connection release

Either user can release the transport connection and end the conversation at any point during data transfer. As mentioned earlier, two forms of connection release are supported by the Transport Layer Interface. The first, abortive release, breaks a connection immediately and can result in the loss of any data that has not yet reached the destination user. `t_snddis` can be called by either user to generate an abortive release. Also, the transport provider can abort a connection if a problem occurs below the Transport Layer Interface. `t_snddis` enables a user to send data to the remote user when aborting a connection. Although the abortive release is supported by all transport providers, the ability to send data when aborting a connection is not.

When the remote user is notified of the aborted connection, `t_rcvdis` must be called to retrieve the disconnect indication. This call returns a reason code that indicates why the connection was aborted, and it returns any user data that may have accompanied the disconnect indication (if the abortive release was initiated by the remote user). This reason code is specific to the underlying transport protocol, and it should not be interpreted by protocol-independent software.

The second form of connection release is orderly release, which gracefully terminates a connection and guarantees that no data is lost. All transport providers must support the abortive release procedure, but orderly release is an optional facility that is not supported by all transport protocols.

The server

The client/server example in this chapter assumes that the transport provider supports the orderly release of a connection. When all the data has been transferred by the server, the connection can be released as follows:

```

        if (t_sndrel(conn_fd) < 0) {
            t_error("t_sndrel failed");
            exit(27);
        }
        pause(); /* until orderly release indication arrives */
    }
}

```

The orderly release procedure consists of two steps by each user. The first user to complete data transfer can initiate a release using **t_sndrel**, as illustrated in the example. This routine informs the client that no more data will be sent by the server. When the client receives such an indication, it can continue sending data back to the server if desired. When all data has been transferred, however, the client must also call **t_sndrel** to indicate that it is ready to release the connection. The connection is released only after both users have requested an orderly release and received the corresponding indication from the other user.

In this example, data is transferred in one direction from the server to the client, and so the server does not expect to receive data from the client after it has initiated the release procedure. Thus, the server simply calls **pause** (see **pause(S)**) after initiating the release. Eventually, the remote user responds with its orderly release request, and the indication generates a signal that will be caught by **connrelease**. Remember that the server earlier issued an **I_SETSIG ioctl** call to generate a signal on any incoming event. Since the only possible Transport Layer Interface events that can occur in this situation are a disconnect indication or an orderly release indication, **connrelease** terminates normally when the orderly release indication arrives. The **exit** call in **connrelease** closes the transport endpoint, thereby freeing the bound address for use by another user. If a user process wants to close a transport endpoint without exiting, it can call **t_close**.

The client

The client's view of connection release is similar to that of the server. As mentioned earlier, the client continues to process incoming data until `t_rcv` fails. If the server releases the connection (using either `t_snddis` or `t_sndrel`), `t_rcv` fails and sets `t_errno` to `TLOOK`. The client then processes the connection release as follows:

```

if ((t_errno == TLOOK) && (t_look(fd) == T_ORDREL)) {
    if (t_rcvrel(fd) < 0) {
        t_error("t_rcvrel failed");
        exit(6);
    }
    if (t_sndrel(fd) < 0) {
        t_error("t_sndrel failed");
        exit(7);
    }
    exit(0);
}
t_error("t_rcv failed");
exit(8);
}

```

Under normal circumstances, the client terminates the transfer of data by calling `t_sndrel` to initiate the connection release. When the orderly release indication arrives at the client's side of the connection, the client checks to make sure that the expected orderly release indication has arrived. If so, it proceeds with the release procedures by calling `t_rcvrel` to process the indication and `t_sndrel` to inform the server that it is also ready to release the connection. At this point the client exits, thereby closing its transport endpoint.

Not all transport providers support the orderly release facility just described, and so users may have to use the abortive release facility provided by `t_snddis` and `t_rcvdis`. However, steps must be taken by each user to prevent any loss of data. For example, a special byte pattern can be inserted in the data stream to indicate the end of a conversation. Many mechanisms are possible for preventing data loss. Each application and high-level protocol must choose an appropriate mechanism given the target protocol environment and requirements.

Connectionless-mode service

This section describes the connectionless-mode service of the Transport Layer Interface. Connectionless-mode service is appropriate for short-term request/response interactions, such as transaction processing applications. Data are transferred in self-contained units with no logical relationship required among multiple units.

The connectionless-mode services are described using a transaction server as an example. This server waits for incoming transaction queries and then processes and responds to each query.

Local management

Like connection-mode service, the transport users must perform appropriate local management steps before data can be transferred. A user must choose the appropriate connectionless service provider using `t_open` and establish its identity using `t_bind`.

`t_optmgmt` can be used to negotiate protocol options that may be associated with the transfer of each data unit. As with the connection-mode service, each transport provider specifies the options, if any, that it supports. Option negotiation is therefore a protocol-specific activity.

In the example, the definitions and local management calls needed by the transaction server are as follows:

```

#include <stdio.h>
#include <fcntl.h>
#include <tiuser.h>

#define SRV_ADDR    2          /* server's well-known address */

main()
{
    int fd;
    int flags;

    struct t_bind *bind;
    struct t_unitdata *ud;
    struct t_uderr *uderr;

    extern int t_errno;

    if ((fd = t_open("/dev/tidg", O_RDWR, NULL)) < 0) {
        t_error("unable to open /dev/provider");
        exit(1);
    }

    if ((bind = (struct t_bind *)t_alloc(fd, T_BIND, T_ADDR)) == NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }

    bind->addr.len = sizeof(int);
    *(int *)bind->addr.buf = SRV_ADDR;
    bind->qlen = 0;

    if (t_bind(fd, bind, bind) < 0) {
        t_error("t_bind failed");
        exit(3);
    }

    /*
     * is the bound address correct?
     */

    if (*(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\n");
        exit(4);
    }
}

```

The local management steps should look familiar by now. The server establishes a transport endpoint with the desired transport provider using `t_open`. Each provider has an associated service type, and so the user can choose a

particular service by opening the appropriate transport provider file. This connectionless-mode server ignores the characteristics of the provider returned by `t_open` in the same way as the users in the connection-mode example, setting the third argument to `NULL`. For simplicity, the transaction server assumes the transport provider has the following characteristics:

- The transport address is an integer value that uniquely identifies each user.
- The transport provider supports the `T_CLTS` service type (connectionless transport service, or datagram).
- The transport provider does not support any protocol-specific options.

The connectionless server also binds a transport address to the endpoint so that potential clients can identify and access the server. A `t_bind` structure is allocated using `t_alloc`, and the `buf` and `len` fields of the address are set accordingly.

One important difference between the connection-mode server and this connectionless-mode server is that the `qlen` field of the `t_bind` structure has no meaning for connectionless-mode service. That is because all users are capable of receiving datagrams once they have bound an address. The Transport Layer Interface defines an inherent client/server relationship between two users while establishing a transport connection in the connection-mode service. However, no such relationship exists in the connectionless-mode service. It is the context of this example, not the Transport Layer Interface, that defines one user as a server and another as a client.

Because the address of the server is known by all potential clients, the server checks the bound address returned by `t_bind` to ensure that it is correct.

Data transfer

Once a user has bound an address to the transport endpoint, datagrams can be sent or received over that endpoint. Each outgoing message is accompanied by the address of the destination user. In addition, the Transport Layer Interface enables a user to specify protocol options that should be associated with the transfer of the data unit (for example, transit delay). As discussed earlier, each transport provider defines the set of options, if any, that can accompany a datagram. When the datagram is passed to the destination user, the associated protocol options can be returned as well.

The following sequence of calls illustrates the data transfer phase of the connectionless-mode server:

```

    if ((ud = (struct t_unitdata *)t_alloc(fd, T_UNITDATA, T_ALL)) == NULL) {
        t_error("t_alloc of t_unitdata structure failed");
        exit(5);
    }
    if ((uderr = (struct t_uderr *)t_alloc(fd, T_UDERROR, T_ALL)) == NULL) {
        t_error("t_alloc of t_uderr structure failed");
        exit(6);
    }
    while (1) {
        if (t_rcvudata(fd, ud, &flags) < 0) {
            if (t_errno == TLOOK) {
                /*
                 * Error on previously sent datagram
                 */
                if (t_rcvuderr(fd, uderr) < 0) {
                    exit(7);
                }
                fprintf(stderr, "bad datagram, error = %d\n",
                    uderr->error);
                continue;
            }

            t_error("t_rcvudata failed");
            exit(8);
        }
        /*
         * Query() processes the request and places the
         * response in ud->udata.buf, setting ud->udata.len
         */
        query(ud);
        if (t_sndudata(fd, ud, 0) < 0) {
            t_error("t_sndudata failed");
            exit(9);
        }
    }
}
query()
{
    /* Merely a stub for simplicity */
}

```

The server must first allocate a `t_unitdata` structure for storing datagrams, which has the following format:

```

struct t_unitdata {
    struct netbuf addr;
    struct netbuf opt;
    struct netbuf udata;
}

```

addr holds the source address of incoming datagrams and the destination address of outgoing datagrams, *opt* identifies any protocol options associated with the transfer of the datagram, and *udata* holds the data itself. The *addr*, *opt*, and *udata* fields must all be allocated with buffers that are large enough to hold any possible incoming values. As described in the previous chapter, the `T_ALL` argument to `t_alloc` ensures this and sets the `maxlen` field of each `netbuf` structure accordingly. Because the provider does not support protocol options in this example, no options buffer is allocated, and `maxlen` is set to zero in the `netbuf` structure for options. A `t_uderr` structure is also allocated by the server for processing any datagram errors, as will be discussed later in this chapter.

The transaction server loops forever, receiving queries, processing the queries, and responding to the clients. It first calls `t_rcvudata` to receive the next query. `t_rcvudata` retrieves the next available incoming datagram. If none is currently available, `t_rcvudata` blocks, waiting for a datagram to arrive. The second argument of `t_rcvudata` identifies the `t_unitdata` structure where the datagram should be stored.

The third argument, *flags*, must point to an integer variable. *flags* can be set to `T_MORE` on return from `t_rcvudata` to indicate that the user's *udata* buffer was not large enough to store the full datagram. In this case, subsequent calls to `t_rcvudata` will retrieve the remainder of the datagram. `t_alloc` allocates a *udata* buffer large enough to store the maximum datagram size, and so the transaction server does not have to check the value of *flags*.

If a datagram is received successfully, the transaction server calls the `query` routine to process the request. This routine stores the response in the structure pointed to by *ud*, and sets *ud->udata.len* to indicate the number of bytes in the response. The source address returned by `t_rcvudata` in *ud->addr* is used as the destination address by `t_sndudata`.

When the response is ready, `t_sndudata` is called to return the response to the client. The Transport Layer Interface prevents a user from flooding the transport provider with datagrams using the same flow control mechanism described for the connection-mode service. In such cases, `t_sndudata` blocks until the flow control is relieved, and then resumes its operation.

Datagram errors

If the transport provider cannot process a datagram that was passed to it by `t_sndudata`, it returns a unit data error event, `T_UDERR`, to the user. This event includes the destination address and options associated with the datagram, plus a protocol-specific error value that describes what may be wrong with the datagram. The reason a datagram could not be processed is protocol-specific. One reason may be that the transport provider could not interpret the destination address or options. Each transport protocol is expected to specify all reasons for which it is unable to process a datagram.

NOTE The unit data error indication is not necessarily intended to indicate success or failure in delivering the datagram to the specified destination. The transport protocol decides how the indication will be used. Remember, the connectionless service does not guarantee reliable delivery of data.

The transaction server is notified of this error event when it attempts to receive another datagram. In this case, `t_rcvudata` fails, setting `t_errno` to `TLOOK`. If `TLOOK` is set, the only possible event is `T_UDERR`, and so the server calls `t_rcvuderr` to retrieve the event. The second argument to `t_rcvuderr` is the `t_uderr` structure which was allocated earlier. This structure is filled in by `t_rcvuderr` and has the following format:

```
struct t_uderr {
    struct netbuf addr;
    struct netbuf opt;
    long error;
}
```

where *addr* and *opt* identify the destination address and protocol options as specified in the bad datagram, and *error* is a protocol-specific error code that indicates why the provider could not process the datagram. The transaction server prints the error code and then continues by entering the processing loop again.

A read/write interface

Sometimes a user may want to establish a transport connection and then `exec` (see `exec(S)`) an existing user program such as `cat` (see `cat(C)`) to process the data as it arrives over the connection. However, existing programs use `read` and `write` for their I/O needs. The Transport Layer Interface does not directly support a `read/write` interface to a transport provider, but one is available with the UNIX System. This interface enables a user to issue `read` and `write` calls over a transport connection that is in the data transfer phase. This section describes the `read/write` interface to the connection-mode service of the Transport Layer Interface. This interface is not available with the connectionless-mode service.

The `read/write` interface is presented using the client example, with some minor modifications, from the section "Connectionless-mode service" earlier in this chapter. The clients are identical until the data transfer phase is reached. At that point, this client uses the `read/write` interface and `cat` to process incoming data. `cat` can be run without change over the transport connection. Only the differences between this client and that of the example in the section "Connectionless-mode service," are shown below.

```
#include <stropts.h>
.
.      /*
.      * Same local management and connection
.      * establishment steps.
.      */
.
.
if (ioctl(fd, I_PUSH, "tirdwr") < 0) {
    perror("I_PUSH of tirdwr failed");
    exit(5);
}

close(0);
dup(fd);
execl("/bin/cat", "/bin/cat", 0);
perror("execl of /bin/cat failed");
exit(6);
}
```

The client invokes the `read/write` interface by pushing the `tirdwr` module (see `tirdwr(S)`) onto the stream associated with the transport endpoint where the connection was established (see `I_PUSH` in `streamio(S)`). This module converts the Transport Layer Interface above the transport provider into a pure `read/write` interface. With the module in place, the client calls `close` (see

`close(S))` and `dup` (see `dup(S)`) to establish the transport endpoint as its standard input file, and uses `/bin/cat` to process the input. Because the transport endpoint identifier is a file descriptor, the facility for **duping** the endpoint is available to users.

Because the Transport Layer Interface has been implemented using STREAMS, the facilities of this character I/O mechanism can be used to provide enhanced user services. By pushing the `tirdwr` module above the transport provider, the user's interface is effectively changed. The semantics of `read` and `write` must be followed, and message boundaries are not preserved.

WARNING The `tirdwr` module can be pushed onto a stream only when the transport endpoint is in the data transfer phase. Once the module is pushed, the user cannot call any Transport Layer Interface routines. If a Transport Layer Interface routine is invoked, `tirdwr` will generate a fatal protocol error, `EPROTO`, on that stream, rendering it unusable. Furthermore, if the user pops the `tirdwr` module off the stream (see `I_POP` in `streamio(S)`), the transport connection will be aborted.

The exact semantics of `write`, `read`, and `close` using `tirdwr` are described below. To summarize, `tirdwr` enables a user to send and receive data over a transport connection using `read` and `write`. This module translates all Transport Layer Interface indications into the appropriate actions. The connection can be released with the `close` system call.

write

The user can transmit data over the transport connection using `write`. The `tirdwr` module passes data through to the transport provider. However, if a user attempts to send a zero-length data packet, which the STREAMS mechanism allows, `tirdwr` will discard the message. If for some reason the transport connection is aborted (for example, if the remote user aborts the connection using `t_snddis`), a STREAMS hangup condition will be generated on that stream, and further `write` calls will fail and set `errno` to `ENXIO`. The user can still retrieve any available data after a hangup, however.

read

read can be used to retrieve data that has arrived over the transport connection. The **tirdwr** module passes data through to the user from the transport provider. However, any other event or indication passed to the user from the provider is processed by **tirdwr** as follows:

- **read** cannot process expedited data because it cannot distinguish expedited data from normal data for the user. If an expedited data indication is received, **tirdwr** generates a fatal protocol error, **EPROTO**, on that stream. This error causes further system calls to fail. Thus, you should not communicate with a process that is sending expedited data.
- If an abortive disconnect indication is received, **tirdwr** discards the indication and generates a **STREAMS** hangup condition on that stream. Subsequent **read** calls retrieve any remaining data, and then **read** returns zero for all further calls (indicating end-of-file).
- If an orderly release indication is received, **tirdwr** discards the indication and delivers a zero-length **STREAMS** message to the user. As described in **read**, this notifies the user of end-of-file by returning 0 to the user.
- If any other Transport Layer Interface indication is received, **tirdwr** generates a fatal protocol error, **EPROTO**, on that stream. This causes further system calls to fail. If a user pushes **tirdwr** onto a stream after the connection has been established, such indications are not generated.

close

With **tirdwr** on a stream, the user can send and receive data over a transport connection for the duration of that connection. Either user can terminate the connection by closing the file descriptor associated with the transport endpoint or by popping the **tirdwr** module off the stream. In either case, **tirdwr** takes the following actions:

- If an orderly release indication was previously received by **tirdwr**, an orderly release request is passed to the transport provider to complete the orderly release of the connection. The remote user who initiated the orderly release procedure receives the expected indication when data transfer completes.
- If a disconnect indication was previously received by **tirdwr**, no special action is taken.
- If neither an orderly release indication nor disconnect indication was previously received by **tirdwr**, a disconnect request is passed to the transport provider to abortively release the connection.

- If an error previously occurred on the stream and a disconnect indication has not been received by `tirdwr`, a disconnect request is passed to the transport provider.

A process cannot initiate an orderly release after `tirdwr` is pushed onto a stream, but `tirdwr` will handle an orderly release properly if it is initiated by the user on the other side of a transport connection. If the client in this chapter is communicating with the server program in the “Connectionless-mode service” section, that server terminates the transfer of data with an orderly release request. The server then waits for the corresponding indication from the client. At that point, the client exits and the transport endpoint is closed. As explained in the first bulleted item above, when the file descriptor is closed, `tirdwr` initiates the orderly release request from the client’s side of the connection. This generates the indication that the server is expecting, and the connection is properly released.

Advanced topics

This section presents important concepts that have not been covered in the previous sections. First, an optional non-blocking (asynchronous) mode for some library calls is described. Then an advanced programming example is presented. This example defines a server which supports multiple outstanding connect indications and operates in an event-driven manner.

Asynchronous execution mode

Many Transport Layer Interface library routines can block waiting for an incoming event or the relaxation of flow control. However, some time-critical applications should not block for any reason. Similarly, an application may wish to do local processing while waiting for some asynchronous Transport Layer Interface event.

Support for asynchronous processing of Transport Layer Interface events is available to applications using a combination of the STREAMS asynchronous features and the non-blocking mode of the Transport Layer Interface library routines. Earlier examples in this chapter have illustrated the use of the STREAMS `poll` system call and the `I_SETSIG` `ioctl` command for processing events in an asynchronous manner.

In addition, each Transport Layer Interface routine that can block waiting for some event can be run in a special non-blocking mode. For example, `t_listen` normally blocks, waiting for a connect indication. However, a server can periodically poll a transport endpoint for existing connect indications by calling `t_listen` in the non-blocking (or asynchronous) mode. The asynchronous mode is enabled by setting `O_NDELAY` on the file descriptor. This can be set

as a flag on `t_open` or by calling `fcntl(S)` before calling the Transport Layer Interface routine. `fcntl` can be used to enable or disable this mode at any time. All programming examples illustrated throughout this chapter use the default, synchronous mode of processing.

`O_NDELAY` affects each Transport Layer Interface routine in a different manner. To determine the exact semantics of `O_NDELAY` for a particular routine, see the appropriate pages in Section S.

Advanced programming example

The following example illustrates two important concepts. The first is a server's ability to manage multiple outstanding connect indications. The second is the ability to write event-driven software using the Transport Layer Interface and the STREAMS system call interface.

The server example in the section "Connection-mode service" is capable of supporting only one outstanding connect indication, but the Transport Layer Interface supports the ability to manage multiple outstanding connect indications. One reason a server might wish to receive several, simultaneous connect indications is to impose a priority scheme on each client. A server can retrieve several connect indications, and then accept them in an order based on a priority associated with each client. A second reason for handling several outstanding connect indications is that the single-threaded scheme has some limitations. Depending on the implementation of the transport provider, it is possible that while the server is processing the current connect indication, other clients will find it busy. However, if multiple connect indications can be processed simultaneously, the server will be found to be busy only if the maximum allowed number of clients attempt to call the server simultaneously.

The server example is event-driven: the process polls a transport endpoint for incoming Transport Layer Interface events and then takes the appropriate actions for the current event. The example demonstrates the ability to poll multiple transport endpoints for incoming events.

The definitions and local management functions needed by this example are similar to those of the server example in the section "Connection-mode service."



Example 8-3 Advanced programming

```

#include <tiuser.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <stropts.h>
#include <signal.h>

#define NUM_FDS          1
#define MAX_CONN_IND 4
#define SRV_ADDR        1      /* server's well-known address */

int conn_fd;                /* server connection here */
struct t_call *calls[NUM_FDS][MAX_CONN_IND]; /* holds connect indications */
extern int t_errno;

main()
{
    struct pollfd pollfds[NUM_FDS];
    struct t_bind *bind;
    int i;

    /*
     * Only opening and binding one transport endpoint,
     * but more could be supported
     */
    if ((pollfds[0].fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open failed");
        exit(1);
    }

    if ((bind = (struct t_bind *)t_alloc(pollfds[0].fd, T_BIND, T_ALL)) == NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }
    bind->qlen = MAX_CONN_IND;
    bind->addr.len = sizeof(int);
    *(int *)bind->addr.buf = SRV_ADDR;
    if (t_bind(pollfds[0].fd, bind, bind) < 0) {
        t_error("t_bind failed");
        exit(3);
    }

    /*
     * Was the correct address bound?
     */
    if (*(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\n");
        exit(4);
    }
}

```

The file descriptor returned by `t_open` is stored in a `pollfd` structure (see `poll(5)`) poll the transport endpoint for incoming data. Notice that only one transport endpoint is established in this example. However, the remainder of the example is written to manage multiple transport endpoints. Several endpoints could be supported with minor changes to the above code.

An important aspect of this server is that it sets `qlen` to a value greater than 1 for `t_bind`. This indicates that the server is willing to handle multiple outstanding connect indications. Remember that the earlier examples single-threaded the connect indications and responses. The server would accept the current connect indication before retrieving additional connect indications. However, this example can retrieve up to `MAX_CONN_IND` connect indications at one time before responding to any of them. The transport provider can negotiate the value of `qlen` downward if it cannot support `MAX_CONN_IND` outstanding connect indications.

Once the server has bound its address and is ready to process incoming connect requests, it does the following:

```

pollfds[0].events = POLLIN;
while (1) {
    if (poll(pollfds, NUM_FDS, -1) < 0) {
        perror("poll failed");
        exit(5);
    }
    for (i = 0; i < NUM_FDS; i++) {
        switch (pollfds[i].revents) {
        default:
            perror("poll returned error event");
            exit(6);
        case 0:
            continue;
        case POLLIN:
            do_event(i, pollfds[i].fd);
            service_conn_ind(i, pollfds[i].fd);
        }
    }
}

```

The `events` field of the `pollfd` structure is set to `POLLIN`, which will notify the server of any incoming Transport Layer Interface events. The server then enters an infinite loop, in which it will poll the transport endpoint(s) for events and then process those events as they occur.

The `poll` call blocks indefinitely, waiting for an incoming event. On return, each entry (corresponding to each transport endpoint) is checked for an existing event. If `revents` is set to 0, no event has occurred on that endpoint. In this case, the server continues to the next transport endpoint. If `revents` is set to `POLLIN`, an event does exist on the endpoint. In this case, `do_event` is called to process the event. If `revents` contains any other value, an error must have occurred on the transport endpoint, and the server exits.

For each iteration of the loop, if any event is found on the transport endpoint, `service_conn_ind` is called to process any outstanding connect indications. However, if another connect indication is pending, `service_conn_ind` saves the current connect indication and responds to it later. This routine will be explained shortly.

If an incoming event is discovered, the following routine is called to process it:

Example 8-4 Processing incoming events (part 1/2)

```
do_event(slot, fd)
{
    struct t_discon *discon;
    int i;

    switch (t_look(fd)) {

    default:
        fprintf(stderr, "t_look returned an unexpected event\n");
        exit(7);

    case T_ERROR:
        fprintf(stderr, "t_look returned T_ERROR event\n");
        exit(8);

    case -1:
        t_error("t_look failed");
        exit(9);

    case 0:
        /* since POLLIN returned, this should not happen */
        fprintf(stderr, "t_look returned no event\n");
        exit(10);
```

Example 8-5 Processing incoming events (2/2)

```

case T_LISTEN:
    /*
     * find free element in calls array
     */
    for (i = 0; i < MAX_CONN_IND; i++) {
        if (calls[slot][i] == NULL)
            break;
    }

    if ((calls[slot][i] = (struct t_call *)t_alloc(fd, T_CALL, T_ALL)) == NULL) {
        t_error("t_alloc of t_call structure failed");
        exit(11);
    }

    if (t_listen(fd, calls[slot][i]) < 0) {
        t_error("t_listen failed");
        exit(12);
    }

    break;

case T_DISCONNECT:
    discon = (struct t_discon *)t_alloc(fd, T_DIS, T_ALL);

    if (t_rcvdis(fd, discon) < 0) {
        t_error("t_rcvdis failed");
        exit(13);
    }

    /*
     * find call ind in array and delete it
     */
    for (i = 0; i < MAX_CONN_IND; i++) {
        if (discon->sequence == calls[slot][i]->sequence) {
            t_free(calls[slot][i], T_CALL);
            calls[slot][i] = NULL;
        }
    }

    t_free(discon, T_DIS);
    break;
}
}

```

This routine takes a number, *slot*, and a file descriptor, *fd*, as arguments. *slot* is used as an index into the global array *calls*. This array contains an entry for each polled transport endpoint, where each entry consists of an array of *t_call* structures that hold incoming connect indications for that transport endpoint. The value of *slot* is used to identify the transport endpoint of interest.

do_event calls **t_look** to determine the Transport Layer Interface event which has occurred on the transport endpoint referenced by *fd*. If a connect indication (**T_LISTEN** event) or disconnect indication (**T_DISCONNECT** event) has arrived, the event is processed. Otherwise, the server prints an appropriate error message and exits.

For connect indications, **do_event** scans the array of outstanding connect indications looking for the first free entry. A **t_call** structure is then allocated for that entry, and the connect indication is retrieved using **t_listen**. There must always be at least one free entry in the connect indication array because the array is large enough to hold the maximum number of outstanding connect indications as negotiated by **t_bind**. The processing of the connect indication is deferred until later.

If a disconnect indication arrives, it must correspond to a previously received connect indication. This scenario arises if a client attempts to undo a previous connect request. In this case, **do_event** allocates a **t_discon** structure to retrieve the relevant disconnect information. This structure has the following members:

```

struct t_discon {
    struct netbuf udata;
    int reason;
    int sequence;
}

```

where *udata* identifies any user data which might have been sent with the disconnect indication, *reason* contains a protocol-specific disconnect reason code, and *sequence* identifies the outstanding connect indication that matches this disconnect indication.

Next, **t_rcvdis** is called to retrieve the disconnect indication. The array of connect indications for *slot* is then scanned for one which contains a sequence number that matches the *sequence* number in the disconnect indication. When the connect indication is found, it is freed and the corresponding entry is set to **NULL**.

As mentioned earlier, if any event is found on a transport endpoint, **service_conn_ind** is called to process all currently outstanding connect indications associated with that endpoint as follows:

```
service_conn_ind(slot, fd)
{
    int i;
    for (i = 0; i < MAX_CONN_IND; i++) {
        if (calls[slot][i] == NULL)
            continue;

        if ((conn_fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
            t_error("open failed");
            exit(14);
        }
        if (t_bind(conn_fd, NULL, NULL) < 0) {
            t_error("t_bind failed");
            exit(15);
        }
        if (t_accept(fd, conn_fd, calls[slot][i]) < 0) {
            if (t_errno == TLOOK) {
                t_close(conn_fd);
                return;
            }
            t_error("t_accept failed");
            exit(16);
        }
        t_free(calls[slot][i], T_CALL);
        calls[slot][i] = NULL;
        run_server(fd);
    }
}
```

For the given slot (the transport endpoint), the array of outstanding connect indications is scanned. For each indication, the server opens a responding transport endpoint, binds an address to the endpoint, and then accepts the connection on that endpoint. If another event (connect indication or disconnect indication) arrives before the current indication is accepted, **t_accept** will fail and set **t_errno** to **TLOOK**.

NOTE The user cannot accept an outstanding connect indication if any pending connect indication events or disconnect indication events exist on that transport endpoint.

If this error occurs, the responding transport endpoint is closed and **service_conn_ind** returns immediately (saving the current connect indication

for later processing). This causes the server's main processing loop to be entered, and the new event is discovered by the next call to poll. In this way, multiple connect indications can be queued by the user.

Eventually all events are processed, and `service_conn_ind` is able to accept each connect indication in turn. Once the connection is established, the `run_server` routine used by the server in the section "Connection-mode service" is called to manage the data transfer.

TLI programming example summary

The examples previously presented throughout this chapter are shown in their entirety in this section.

Connection-mode client

The following code is for the connection-mode client program described in the section "Connection-mode service". This client establishes a transport connection with a server and then receives data from the server and writes it to its standard output. The connection is released using the orderly release facility of the Transport Layer Interface. This client will communicate with each of the connection-mode servers presented in the chapter.

Example 8-6 Connection-mode client example (1/2)

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

#define SRV_ADDR    1    /* server's well-known address */

main()
{
    int fd;
    int nbytes;
    int flags = 0;
    char buf[1024];
    struct t_call *sndcall;
    extern int t_errno;

    if ((fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open failed");
        exit(1);
    }
}
```

Example 8-7 Connection-mode client example (2/2)

```

    if (t_bind(fd, NULL, NULL) < 0) {
        t_error("t_bind failed");
        exit(2);
    }

    /*
     * By assuming that the address is an integer value,
     * this program may not run over another protocol.
     */
    if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL) {
        t_error("t_alloc failed");
        exit(3);
    }
    sndcall->addr.len = sizeof(int);
    *(int *)sndcall->addr.buf = SRV_ADDR;

    if (t_connect(fd, sndcall, NULL) < 0) {
        t_error("t_connect failed for fd");
        exit(4);
    }

    while ((nbytes = t_rcv(fd, buf, 1024, &flags)) != -1) {
        if (fwrite(buf, 1, nbytes, stdout) < 0) {
            fprintf(stderr, "fwrite failed\n");
            exit(5);
        }
    }

    if ((t_errno == TLOOK) && (t_look(fd) == T_ORDREL)) {
        if (t_rcvrel(fd) < 0) {
            t_error("t_rcvrel failed");
            exit(6);
        }
        if (t_sndrel(fd) < 0) {
            t_error("t_sndrel failed");
            exit(7);
        }
        exit(0);
    }
    t_error("t_rcv failed");
    exit(8);
}

```

Connection-mode server

The following code is for the connection-mode server program described in the section “Connection-mode service.” This server establishes a transport connection with a client and then transfers a log file to the client on the other side of the connection. The connection is released using the orderly release facility of the Transport Layer Interface. The connection-mode client presented earlier will communicate with this server.

Example 8-8 Connection-mode server example (1/4)

```
#include <tiuser.h>
#include <stropts.h>
#include <fcntl.h>
#include <stdio.h>
#include <signal.h>

#define DISCONNECT -1
#define SRV_ADDR    1 /* server's well-known address */

int conn_fd;          /* connection established here */
extern int t_errno;

main()
{
    int listen_fd;      /* listening transport endpoint */
    struct t_bind *bind;
    struct t_call *call;

    if ((listen_fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open failed for listen_fd");
        exit(1);
    }

    /*
     * By assuming that the address is an integer value,
     * this program may not run over another protocol.
     */
    if ((bind = (struct t_bind *)t_alloc(listen_fd, T_BIND, T_ALL)) == NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }
    bind->qlen = 1;
    bind->addr.len = sizeof(int);
    *(int *)bind->addr.buf = SRV_ADDR;
```

Example 8-9 Connection-mode server example (2/4)

```

    if (t_bind(listen_fd, bind, bind) < 0) {
        t_error("t_bind failed for listen_fd");
        exit(3);
    }

    /*
     * Was the correct address bound?
     */
    if (*(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\n");
        exit(4);
    }

    if ((call = (struct t_call *)t_alloc(listen_fd, T_CALL, T_ALL)) == NULL) {
        t_error("t_alloc of t_call structure failed");
        exit(5);
    }

    while (1) {
        if (t_listen(listen_fd, call) < 0) {
            t_error("t_listen failed for listen_fd");
            exit(6);
        }

        if ((conn_fd = accept_call(listen_fd, call)) != DISCONNECT)
            run_server(listen_fd);
    }

    accept_call(listen_fd, call)
    int listen_fd;
    struct t_call *call;
    {
        int resfd;

        if ((resfd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
            t_error("t_open for responding fd failed");
            exit(7);
        }

        if (t_bind(resfd, NULL, NULL) < 0) {
            t_error("t_bind for responding fd failed");
            exit(8);
        }
    }

```

Example 8-10 Connection-mode server example (3/4)

```

    if (t_accept(listen_fd, resfd, call) < 0) {
        if (t_errno == TLOOK) { /* must be a disconnect */
            if (t_rcvdis(listen_fd, NULL) < 0) {
                t_error("t_rcvdis failed for listen_fd");
                exit(9);
            }
            if (t_close(resfd) < 0) {
                t_error("t_close failed for responding fd");
                exit(10);
            }
            /* go back up and listen for other calls */
            return(DISCONNECT);
        }
        t_error("t_accept failed");
        exit(11);
    }
    return(resfd);
}

connrelease()
{
    /* conn_fd is global because needed here */
    if (t_look(conn_fd) == T_DISCONNECT) {
        fprintf(stderr, "connection aborted\n");
        exit(12);
    }

    /* else orderly release indication - normal exit */
    exit(0);
}

run_server(listen_fd)
int listen_fd;
{
    int nbytes;
    FILE *logfp;          /* file pointer to log file */
    char buf[1024];
    switch (fork()) {
    case -1:
        perror("fork failed");
        exit(20);

```

Example 8-11 Connection-mode server example (4/4)

```

default:    /* parent */

    /* close conn_fd and then go up and listen again */
    if (t_close(conn_fd) < 0) {
        t_error("t_close failed for conn_fd");
        exit(21);
    }
    return;
case 0:     /* child */

    /* close listen_fd and do service */
    if (t_close(listen_fd) < 0) {
        t_error("t_close failed for listen_fd");
        exit(22);
    }
    if ((logfp = fopen("logfile", "r")) == NULL) {
        perror("cannot open logfile");
        exit(23);
    }

    signal(SIGPOLL, connrelease);
    if (ioctl(conn_fd, I_SETSIG, S_INPUT) < 0) {
        perror("ioctl I_SETSIG failed");
        exit(24);
    }
    if (t_look(conn_fd) != 0) { /* was disconnect already there? */
        fprintf(stderr, "t_look returned unexpected event\n");
        exit(25);
    }

    while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
        if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
            t_error("t_snd failed");
            exit(26);
        }

    if (t_sndrel(conn_fd) < 0) {
        t_error("t_sndrel failed");
        exit(27);
    }
    pause(); /* until orderly release indication arrives */
}
}

```

Connectionless-mode transaction server

The following code is for the connectionless-mode transaction server program described in the section “Connectionless-mode service.” This server waits for incoming datagram queries and then processes each query and sends a response.

Example 8-12 Connectionless transaction server example (1/2)

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

#define SRV_ADDR 2          /* server's well-known address */

main()
(
    int fd;
    int flags;
    struct t_bind *bind;
    struct t_unitdata *ud;
    struct t_uderr *uderr;
    extern int t_errno;

    if ((fd = t_open("/dev/tidg", O_RDWR, NULL)) < 0) {
        t_error("unable to open /dev/provider");
        exit(1);
    }
    if ((bind = (struct t_bind *)t_alloc(fd, T_BIND, T_ADDR)) == NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }
    bind->addr.len = sizeof(int);
    *(int *)bind->addr.buf = SRV_ADDR;
    bind->qlen = 0;

    if (t_bind(fd, bind, bind) < 0) {
        t_error("t_bind failed");
        exit(3);
    }
    /* is the bound address correct? */

    if (*(int *)bind->addr.buf != SRV_ADDR) {
        fprintf(stderr, "t_bind bound wrong address\n");
        exit(4);
    }
}
```

Example 8-13 Connectionless transaction server example (2/2)

```

    if ((ud = (struct t_unitdata *)t_alloc(fd, T_UNITDATA, T_ALL)) == NULL) {
        t_error("t_alloc of t_unitdata structure failed");
        exit(5);
    }
    if ((uderr = (struct t_uderr *)t_alloc(fd, T_UDERROR, T_ALL)) == NULL) {
        t_error("t_alloc of t_uderr structure failed");
        exit(6);
    }
    while (1) {
        if (t_rcvudata(fd, ud, &flags) < 0) {
            if (t_errno == TLOOK) {
                /* Error on previously sent datagram */
                if (t_rcvuderr(fd, uderr) < 0) {
                    t_error("t_rcvuderr failed");
                    exit(7);
                }
                fprintf(stderr, "bad datagram, error = %d\n",
                    uderr->error);
                continue;
            }
            t_error("t_rcvudata failed");
            exit(8);
        }

        /* Query() processes the request and places the
        response in ud->udata.buf, setting ud->udata.len */
        query(ud);

        if (t_sndudata(fd, ud, 0) < 0) {
            t_error("t_sndudata failed");
            exit(9);
        }
    }
}

query()
{
    /* Merely a stub for simplicity */
}

```

Read/write client

The following code represents the connection-mode **read/write** client program described in the section, “A read/write interface,” earlier in this chapter. This client establishes a transport connection with a server. It then uses **cat** to retrieve the data sent by the server and write it to its standard output. This client will communicate with each of the connection-mode servers presented in the chapter.

Example 8-14 Read/write client example

```

#include <stdio.h>
#include <tiuser.h>
#include <fcntl.h>
#include <stropts.h>

#define SRV_ADDR 1 /* server's well-known address */

main()
(
    int fd;
    int nbytes;
    int flags = 0;
    char buf[1024];
    struct t_call *sndcall;
    extern int t_errno;

    if ((fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open failed");
        exit(1);
    }
    if (t_bind(fd, NULL, NULL) < 0) {
        t_error("t_bind failed");
        exit(2);
    }
    /* By assuming that the address is an integer value,
       this program may not run over another protocol. */
    if ((sndcall = (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL) {
        t_error("t_alloc failed");
        exit(3);
    }
    sndcall->addr.len = sizeof(int);
    *(int *)sndcall->addr.buf = SRV_ADDR;
    if (t_connect(fd, sndcall, NULL) < 0) {
        t_error("t_connect failed for fd");
        exit(4);
    }
    if (ioctl(fd, I_PUSH, "tirdwr") < 0) {
        perror("I_PUSH of tirdwr failed");
        exit(5);
    }
    close(0);
    dup(fd);
    execl("/bin/cat", "/bin/cat", 0);
    perror("execl of /bin/cat failed");
    exit(6);
}

```

Event-driven server

The following code represents the connection-mode server program described in the section “Advanced topics.” This server manages multiple connect indications in an event-driven manner. Either connection-mode client presented earlier will communicate with this server.

Example 8-15 Event-driven server example (1/6)

```
#include <tiuser.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <stropts.h>
#include <signal.h>

#define NUM_FDS      1
#define MAX_CONN_IND  4
#define SRV_ADDR      1                                /* server's well-known address */

int conn_fd;                                           /* server connection here */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];         /* holds connect indications */
extern int t_errno;

main()
{
    struct pollfd pollfds[NUM_FDS];
    struct t_bind *bind;
    int i;

    /* Only opening and binding one transport endpoint,
     * but more could be supported */
    if ((pollfds[0].fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
        t_error("t_open failed");
        exit(1);
    }
    if ((bind = (struct t_bind *)t_alloc(pollfds[0].fd, T_BIND, T_ALL)) == NULL) {
        t_error("t_alloc of t_bind structure failed");
        exit(2);
    }
    bind->qlen = MAX_CONN_IND;
    bind->addr.len = sizeof(int);
    *(int *)bind->addr.buf = SRV_ADDR;

    if (t_bind(pollfds[0].fd, bind, bind) < 0) {
        t_error("t_bind failed");
        exit(3);
    }
}
```

Example 8-16 Event-driven server example (2/6)

```

/* Was the correct address bound? */
if (*(int *)bind->addr.buf != SRV_ADDR) {
    fprintf(stderr, "t_bind bound wrong address\n");
    exit(4);
}
pollfds[0].events = POLLIN;
while (1) {
    if (poll(pollfds, NUM_FDS, -1) < 0) {
        perror("poll failed");
        exit(5);
    }
    for (i = 0; i < NUM_FDS; i++) {

        switch (pollfds[i].revents) {

        default:
            perror("poll returned error event");
            exit(6);

        case 0:
            continue;

        case POLLIN:
            do_event(i, pollfds[i].fd);
            service_conn_ind(i, pollfds[i].fd);

        }

    }
}
do_event(slot, fd)
{
    struct t_discon *discon;
    int i;

    switch (t_look(fd)) {

    default:
        fprintf(stderr, "t_look returned an unexpected event\n");
        exit(7);

    case T_ERROR:
        fprintf(stderr, "t_look returned T_ERROR event\n");
        exit(8);

```

Example 8-17 Event-driven server example (3/6)

```

case -1:
    t_error("t_look failed");
    exit(9);

case 0:
    /* since POLLIN returned, this should not happen */
    fprintf(stderr, "t_look returned no event\n");
    exit(10);

case T_LISTEN:
    /* find free element in calls array */
    for (i = 0; i < MAX_CONN_IND; i++) {
        if (calls[slot][i] == NULL)
            break;
    }
    if ((calls[slot][i] = (struct t_call *)t_alloc(fd, T_CALL, T_ALL)) == NULL)
    {
        t_error("t_alloc of t_call structure failed");
        exit(11);
    }
    if (t_listen(fd, calls[slot][i]) < 0) {
        t_error("t_listen failed");
        exit(12);
    }
    break;

case T_DISCONNECT:
    discon = (struct t_discon *)t_alloc(fd, T_DIS, T_ALL);

    if (t_rcvdis(fd, discon) < 0) {
        t_error("t_rcvdis failed");
        exit(13);
    }
    /* find call ind in array and delete it */
    for (i = 0; i < MAX_CONN_IND; i++) {
        if (discon->sequence == calls[slot][i]->sequence) {
            t_free(calls[slot][i], T_CALL);
            calls[slot][i] = NULL;
        }
    }
    t_free(discon, T_DIS);
    break;
}
}

```

Example 8-18 Event-driven server example (4/6)

```

service_conn_ind(slot, fd)
{
    int i;

    for (i = 0; i < MAX_CONN_IND; i++) {
        if (calls[slot][i] == NULL)
            continue;

        if ((conn_fd = t_open("/dev/tivc", O_RDWR, NULL)) < 0) {
            t_error("open failed");
            exit(14);
        }
        if (t_bind(conn_fd, NULL, NULL) < 0) {
            t_error("t_bind failed");
            exit(15);
        }
        if (t_accept(fd, conn_fd, calls[slot][i]) < 0) {
            if (t_errno == TLOOK) {
                t_close(conn_fd);
                return;
            }
            t_error("t_accept failed");
            exit(16);
        }
        t_free(calls[slot][i], T_CALL);
        calls[slot][i] = NULL;

        run_server(fd);
    }
}

connrelease()
{
    /* conn_fd is global because needed here */
    if (t_look(conn_fd) == T_DISCONNECT) {
        fprintf(stderr, "connection aborted\n");
        exit(12);
    }
    /* else orderly release indication - normal exit */
    exit(0);
}

```

Example 8-19 Event-driven server example (5/6)

```

run_server(listen_fd)
int listen_fd;
{
    int nbytes;
    FILE *logfp;           /* file pointer to log file */
    char buf[1024];

    switch (fork()) {
    case -1:
        perror("fork failed");
        exit(20);

    default: /* parent */

        /* close conn_fd and then go up and listen again */
        if (t_close(conn_fd) < 0) {
            t_error("t_close failed for conn_fd");
            exit(21);
        }
        return;

    case 0: /* child */

        /* close listen_fd and do service */
        if (t_close(listen_fd) < 0) {
            t_error("t_close failed for listen_fd");
            exit(22);
        }
        if ((logfp = fopen("logfile", "r")) == NULL) {
            perror("cannot open logfile");
            exit(23);
        }
        signal(SIGPOL, connrelease);
        if (ioctl(conn_fd, I_SETSIG, S_INPUT) < 0) {
            perror("ioctl I_SETSIG failed");
            exit(24);
        }
        if (t_look(conn_fd) != 0) { /* was disconnect already there? */
            fprintf(stderr, "t_look returned unexpected event\n");
            exit(25);
        }
    }
}

```

Example 8-20 Event-driven server example (6/6)

```
        while ((nbytes = fread(buf, 1, 1024, logfp)) > 0)
            if (t_snd(conn_fd, buf, nbytes, 0) < 0) {
                t_error("t_snd failed");
                exit(26);
            }
        if (t_sndrel(conn_fd) < 0) {
            t_error("t_sndrel failed");
            exit(27);
        }
        pause();      /* until orderly release indication arrives */
    }
}
```



Index

A

Abortive release, 172
 Address space, 158
 Alignment *See* Storage alignment
 Arguments, variable number of, 162
 Assembly language
 interface, 149
 return values, 146
 routines, 146, 149
 Assembly-language interface, described, 141
 Asynchronous execution mode, 206
 audit.h header file, 14

B

Base, 131
 Bit fields, 155
 Byte length, 152
 Byte order, 154, 166

C

C calling conventions, described, 141
 C language
 calling sequence, 145
 interface with assembly language, 149
 return values, 146
 C2 security, 13
 Call sequence, 141
 Calling sequence
 C language, 145
 assembly language, 145
 Canonic Frame, 130
 Character
 classification, macros, 158
 set, 158
 types, 159
 Class name, LSEG, 130
 Client program
 connection release, 196
 data transfer, 191
 definitions, 177
 establishing a connection, 183
 Combination attribute, 96

Comment, record, 114
 Comment, 114
 Compiler, portability problems, 159
 Complete name, LSEG, 130
 Connection establishment, 171, 182
 Connection release, 172, 194
 Connectionless-mode
 defined, 169
 phases, 173
 transaction server, 220
 Connectionless-mode service
 data transfer, 199
 datagram errors, 202
 local management, 197
 Connection-mode
 client, example, 214
 defined, 168
 discussion, 174
 event-driven server, example, 224
 phases of, 169
 server, 216
 ctype.h macros, 158
 curses(S), running under ksh job control, 8

D

DAC (Discretionary Access Control), 16
 Data
 portability, 164
 types, size of, 153
 Data structures, x.out symbol table, 124
 Data transfer, 172-173, 189
 data types, ISAM file, 71
 database management system
 ISAM, 63
 Datagram errors, 202
 Deleting, record, 79

E

EAX register, 146, 149
 EBP register, 146, 149
 EBX register, 149
 ECX register, 149

EDI register, 146, 149
 EDX register, 146, 149
 Eight-leaf descriptor, 99
 encryption, 48
 Environment, portability problems, 163
 error handling, ISAM, 70
 ESI register, 146, 149
 ESP register, 146
 Establishing a connection, 182
 Evaluation order, 162
 Event handling, 184
 Executable format, 124

F

field, in ISAM file, 64
 file, ISAM, 63
 file and record locking
 ISAM, 68
 FIXUP, record, 107
 FIXUPP, 107
 Fixups
 definition, 130
 segment-relative, 132, 137
 self-relative, 132, 138
 Frame
 definition, 131
 specifying, 134
 Frame number, 132
 Functions, arguments, variable number of,
 162

G

Group, 132
 Group definition record, 98
 GRPDEF, 98

H

Hardware reference numbers, 129
 Hibble, 131

I

Identifier length *See* Names, length
 Implicit bss, 123
 Include files, portability problems, 152
 index, in ISAM file, 65

Index fields, 132
 Indices, 132
 ISAM files
 See also isaddindex(S)
 See also isclose(S)
 See also isdelete(S)
 See also isdelindex(S)
 See also isopen(S)
 See also isread(S)
 See also isrewcurr(S)
 See also isstart(S)
 See also iswcurr(S)
 See also iswrite(S)
 ISAM files, 63
 Iterated segments, 122

J

Job control, ksh, 7

K

Korn shell (ksh) 7
 ksh, job control, 7

L

Large model, 123
 ldint(), 73
 ldlong(), 73
 LIDATA, 106
 Line numbers record, 104
 LINNUM, 104
 List of names record, 93
 LNNAMES, 93
 LOBYTE, 131
 Local channel, 175
 Local management, 173, 175
 Local management phase, 169
 Location, types, 131
 Logical iterated data record, 106
 Logical Segment, 136
 LSEG, 136

M

Machine-format floating point, 73
 Macros, character classification, 158
 MAS, 136
 Memory Address Space, 136
 Mode, 132
 MODEND, 112
 Module, 136
 Module end record, 112
 Module header record, 89

N

Names, length, 160
 Network Services Library, 168
 Non-iterated segments, 123
 Numeric record types, 90

O

Object file format, executable, 119
 Object Module Formats, 87, 136
 Offset, 131
 OMF, 136
 omf Subset, 120
 Open Systems Interconnection (OSI), 167
 Orderly release facility, 173
 OSI Reference Model, explanation, 167
 Overlay Name, LSEG, 136

P

Paragraph number, 136
 Path names, portability problems, 152
 Physical Segment, 136
 Pointers, manipulation, 156
 Portability
 address space, 158
 bit fields, 155
 byte length, 152
 byte order, 154, 166
 case distinctions, 160
 character set, 158
 data, 164
 data types, size of, 153
 environment differences, 163
 evaluation order, 162
 functions with variable number of
 arguments, 162

Portability (*continued*)

 guidelines, 152
 hardware, 152
 identifier length, 160
 include files, 152
 path names, 152
 pointer manipulation, 156
 register variables, 160
 shift operations, 159
 side effects, 162
 sign extension, 159
 signed and unsigned char types, 159
 storage alignment, 153
 type conversion, 160
 word length, 153
 Protected subsystem, definition, 14
 prot.h header file, 14
 Protocol options, negotiating, 176
 PSEG
 definition, 136
 number, 136
 PUBDEF, 101
 Public names definition record, 101

R

Random access, 77
 Read/write client, example, 222
 read/write interface, 203
 record, in ISAM file, 64
 Record format, sample, 91
 Record formats, 87
 record locking, ISAM, 68
 Record order, 88
 Record types, numeric, 90
 Record types, 117
 Register variables, 160
 Registers
 EAX, 146, 149
 EBP, 146, 149
 EBX, 149
 ECX, 149
 EDI, 146, 149
 EDX, 146, 149
 ESI, 146, 149
 ESP, 146
 relocatable memory images, 87
 Return values, assembly language, 146
 Return values, 143
 Routine entry sequence, 142

Routine exit sequence, 143
 Routines
 assembly language, 146, 149

S

Sample x.out File, 121
 SecureWare, 14
 security.h header file, 14
 SEGDEF, 94
 Segment addressing, 88
 Segment definition, 88
 Segment definition record, 94
 Segment Name, LSEG, 137
 Segment numbers, 129
 Segment registers, 149
 Segment-relative fixups, 132, 137
 Self-relative fixups, 132, 138
 Server program
 connection establishment, 185
 connection release, 195
 data transfer, 192
 definitions, 179
 Service types, 175
 Seven layer model, 167
 Shift operations, 159
 Side effects, 162
 Sign extension, 159
 Signal handling, code example, 8
 signal handling, security implications, 14
 Signed char type, 159
 Special header fields, 123
 Stack order, 141
 State tables, 174
 State transitions, 174
 Storage alignment, 153
 Symbol definition, 138
 Symbol table, 124

T

target, 133
 TCB (Trusted Computing Base), 16
 T_CLTS service type, 175
 T_CONNECT event, 184
 T_COTS service type, 175
 T_COTS_ORD service type, 175
 T_DATA event, 184
 T_DISCONNECT event, 184
 T_EXDATA event, 184

THEADR, 92
 tirdwr module, example, 203
 TLI, overview, 167
 T_LISTEN event, 184
 TLOOK error, 184
 T-module, 139
 T-module Header Record (THEADR), 92
 T_MORE flag, 190
 t_open routine, 169
 t_optmngmt facility, 176
 T_ORDREL event, 184
 Transport address, 170
 Transport endpoint, 169, 175
 Transport Layer Interface
 local management routines, 170
 overview, 167
 Transport provider, choosing, 169
 Transport provider, 168
 Transport user, 168
 t_rcv, 189
 Trusted facilities
 See also audit(HW)
 See also audit(S)
 See also authaudit(S)
 See also authcap(S)
 See also identity(S)
 code examples, 18, 23, 41
 Trusted facilities, 13
 TSU size, 190
 t_snd, 189
 T_UDERR event, 184
 TYPDEF, 99
 Types, conversion, 160

U

Unsigned char type, 159
 Updating, 78

V

Variables, register *See* Register variables
 Virtual circuit, 171

W

Word boundaries, 73

X

x.out

- file layout, 121

- general description, 120

- implicit bss, 123

- iterated segments, 122

- large model, 123

- non-iterated segments, 123

- special fields, 123

- symbol table, 124

x.out examples, 125

x.out Executable Format, 124

x.out format, 119

x.out include files, 125

x.out segmented OMF specification, 119



Please help us to write computer manuals that meet your needs by completing this form. Please post the completed form to the Technical Publications Research Coordinator nearest you: The Santa Cruz Operation, Ltd., Croxley Centre, Hatters Lane, Watford WD1 8YN, United Kingdom; The Santa Cruz Operation, Inc., 400 Encinal Street, P.O. Box 1900, Santa Cruz, California 95061, USA or SCO Canada, Inc., 130 Bloor Street West, 10th Floor, Toronto, Ontario, Canada M5S 1N5.

Volume title: _____

(Copy this from the title page of the manual, for example, SCO UNIX Operating System User's Guide)

Product: _____

(example, SCO UNIX System V Release 3.2 Operating System Version 4.0)

How long have you used this product?

- ☐ Less than one month ☐ Less than six months ☐ Less than one year
☐ 1 to 2 years ☐ More than 2 years

How much have you read of this manual?

- ☐ Entire manual ☐ Specific chapters ☐ Used only for reference

	Agree		Disagree	
The software was fully and accurately described	☐	☐	☐	☐
The manual was well organized	☐	☐	☐	☐
The writing was at an appropriate technical level (neither too complicated nor too simple)	☐	☐	☐	☐
It was easy to find the information I was looking for	☐	☐	☐	☐
Examples were clear and easy to follow	☐	☐	☐	☐
Illustrations added to my understanding of the software	☐	☐	☐	☐
I liked the page design of the manual	☐	☐	☐	☐

If you have specific comments or if you have found specific inaccuracies, please report these on the back of this form or on a separate sheet of paper. In the case of inaccuracies, please list the relevant page number.

May we contact you further about how to improve SCO UNIX documentation? If so, please supply the following details:

Name _____ Position _____

Company _____

Address _____

City & Post/Zip Code _____

Country _____

Telephone _____ Facsimile _____

10 December 1991



BH01 403P000

58085



AU01407P000