INTERACTIVE

p r o d u c t   f a m i l y

SunSoft
A Sun Microsystems, Inc. Business

AT&T

UNIX® System V/386
Release 3.2

Programmer's Guide
Volume II

Table of Contents

4   awk

5   lex

6   yacc

7 File and Record Locking

8 Shared Libraries

9 Interprocess Communication

Volume II

10 Extended Terminal Interface

11   Common Object File Format (coff)

12   The Link Editor

13   make

14   Source Code Control System (sccs)

15 sdb—the Symbolic Debugger

16 lint

17 C Language

List of Figures

Introduction

Purpose

This guide is designed to give you information about programming in the UNIX System V/386 operating system environment. It does not attempt to teach you how to write programs. Rather, it is intended to supplement texts on programming languages by concentrating on the other elements that are part of getting programs into operation.

Audience and Prerequisite Knowledge

As the title suggests, we are addressing programmers, especially those who have not worked extensively with the UNIX System. No special level of programming involvement is assumed. We hope the book will be useful to people who write only an occasional program, as well as to those who work on or manage large application development projects.

Programmers in the expert class, or those engaged in developing system software, may find this guide lacks the depth of information they need. For them we recommend the *Programmer's Reference Manual*.

Knowledge of terminal use, of a UNIX System editor, and of the UNIX System directory/file structure is assumed. If you feel shaky about your mastery of these basic tools, you might want to look over the *User's Guide* before tackling this one. The material is organized into eighteen chapters.

The C Connection

The UNIX System supports many programming languages, and C compilers are available on many different operating systems. Nevertheless, the relationship between the UNIX Operating System and C has always been and remains very close. Most of the code in the UNIX Operating System is C, and over the years many organizations using the UNIX System have come to use C for an increasing portion of their application code. Thus, while this guide is intended to be useful to you no matter what language(s) you are using, you will find that, unless there is a specific language-dependent point to be made, the examples assume you are programming in C.

Hardware/Software Dependencies

The text reflects the way things work on your computer running UNIX System V/386 at the Release 3.2 level. If you find commands that work a little differently in your UNIX System environment, it may be because you are running under a different release of the software. If some commands do not seem to exist at all, they may be members of packages not installed on your system. Appendix A describes the command packages available on your computer. If you do find yourself trying to execute a non-existent command, check Appendix A, then talk to the administrators of your system.

Notational Conventions

Whenever the text includes examples of output from the computer and/or commands entered by you, we follow the standard notation scheme that is common throughout UNIX System documentation:

■ Commands, options, arguments to commands, and directory and file names that you type in from your terminal are shown in **bold** type.

■ Text that is printed on your terminal by the computer is shown in constant width type. Constant width type is also used for code samples because it allows the most accurate representation of spacing. Spacing is often a matter of coding style but is sometimes critical.

■ Comments added to a display to show that part of the display has been omitted are shown in *italic* type and are indented to separate them from the text that represents computer output or input. Comments that explain the input or output are shown in the same type font as the rest of the display.

Italics are also used to show substitutable values, such as, *filename*, when the format of a command is shown.

■ There is an implied RETURN at the end of each command and menu response you enter. Where you may be expected to enter only a RETURN (as in the case where you are accepting a menu default), the symbol <**CR**> is used.

- In cases where you are expected to enter a control character, it is shown as, for example, **CTRL-D**. This means that you press the **d** key on your keyboard while holding down the **CTRL** key.

- The dollar sign (**$**) and pound sign (**#**) are the standard default prompt signs for an ordinary user and **root**. **$** means you are logged in as an ordinary user. **#** means you are logged in as **root**.

- When the **#** prompt is used in an example, it means the command illustrated may be used only by **root**.

Command References

When commands are mentioned in a section of the text for the first time, a reference to the manual section where the command is formally described is included in parentheses, that is, **command**(section). Numbered sections are located in the following manuals:

Sections (1, 1M), (7)   *User's/System Administrator's Reference Manual*

Sections (1), (2), (3), (4), (5) *Programmer's Reference Manual*

Information in the Examples

While every effort has been made to present displays of information just as they appear on your terminal, it is possible that your system may produce slightly different output. Some displays depend on a particular machine configuration that may differ from yours. Changes between releases of the UNIX System software may cause small differences in what appears on your terminal.

Where complete code samples are shown, we have tried to make sure they compile and work as represented. Where code fragments are shown, while we cannot say that they have been compiled, we have attempted to maintain the same standards of coding accuracy for them.

10 Extended Terminal Interface

Windows 10-58

Panels 10-69

Compiling and Linking Panel Programs 10-70

Creating Panels 10-71

Setting the Field Foreground, Background, and Pad Character

Some Helpful Features of Fields

Manipulating Field Options

Creating and Freeing Forms

Manipulating Form Attributes

Displaying Forms

TAM Transition Library

Compiling and Running TAM Applications under ETI

Tips for Polishing TAM Application Programs Running under ETI

How the TAM Transition Library Works

Program Examples

Overview

Screen management programs are a common component of many commercial computer applications. These programs handle input and output at a video display terminal. A screen program might move a cursor, print a display, divide a terminal screen into windows, or change the definition of colors. Many screen management programs build end-user terminal interfaces to help users enter and retrieve information from a database — interfaces such as forms, menus, and help and error message displays.

This chapter explains how to use the Extended Terminal Interface (ETI) package to write screen management programs on a UNIX System V/386 system. (It also tells you what you need to know about the **terminfo** database to use ETI.) To start you writing screen management programs as soon as possible, the information in this chapter does not cover every routine in the libraries. Although it covers all routines in the high-level libraries (those that build panels, menus, and forms), it covers only the most frequently used routines in the low-level library (**curses**). For more information, this chapter points you to the **curses**(3X), **terminfo**(4), and other manual pages in the *Programmer's Reference Manual*. Keep this manual close at hand; you'll find it invaluable when you want to know more about these and other routines.

Because the routines are compiled C functions, you should be familiar with the C programming language before using ETI. You should also be familiar with the UNIX System/C language standard I/O package (see "System Calls and Subroutines" and "Input/Output" in Chapter 2 and the **stdio**(3S) manual page of the *Programmer's Reference Manual*). With that knowledge and an appreciation for the UNIX System philosophy of building on the work of others, you can design screen management programs for many purposes.

How this Chapter is Organized

This chapter contains eleven sections:

■ Introduction to ETI

This is the present section. It briefly describes the ETI libraries and how ETI works with the **terminfo** database.

■ Basic ETI Programming

This section describes the routines and other components that every ETI program needs to work properly, tells you how to compile and run

low-level ETI **(curses)** programs, and introduces important concepts such as refreshing.

■ Simple Input and Output

This section describes the routines in the low-level ETI **(curses)** library for writing to, and reading from, a screen and manipulating colors. It also covers the suite of video attributes and options which enable you to enhance your displays with striking visual effects.

■ Windows

This section explains the use of windows and subwindows. It delves more deeply into the refresh operation and covers the functions **wnoutrefresh()** and **doupdate()**.

■ Panels

This section begins the treatment of the high-level ETI functions. It describes the use of panels—windows with interrelationships of depth—and covers the set of panel functions, which enable you to create panels, move them, associate them with different windows, place them on top of other panels, and so forth.

■ Menus

This section explains the suite of menu functions. It explains how to create menu items and menus, display them, change menu video attributes, have users interact with menus, and more.

■ Forms

This section covers the wealth of form functions. It shows how to create fields and forms, display them, change form video attributes, have users interact with forms, and more.

■ Other ETI Routines

This section covers routines for screen management programs that draw line graphics, use a terminal's soft labels, and work with more than one terminal at the same time.

■ Working with **terminfo** Routines

This section describes a subset of routines in the **curses** library. These routines access and manipulate data in the **terminfo** database. They are used to set up and handle special terminal capabilities such as programmable function keys. This section also describes the **terminfo**

database, related support tools, and their relationship to the **curses** library.

■ Terminal Access Method (TAM) Transition Library

This section explains how to use the TAM transition library and how to rewrite TAM application programs to run efficiently under ETI without the TAM transition library.

■ Program Examples

This section includes programs that illustrate uses of low level ETI **curses** routines.

Conventions Used in this Chapter

This section uses the following conventions to discuss ETI routines:

■ In program text, the major ETI data types appear in uppercase. They are as follows:

☐ WINDOW—a rectangular area of the screen treated as a unit

☐ PANEL—a window with relations of depth to other windows so that regions hidden behind other windows are invisible

☐ ITEM—a character string consisting of a name and an optional description

☐ MENU—a screen display that presents a set of items from which the user chooses one or more, depending on the type of menu

☐ FIELD—an m x n block of character positions within a form that ETI functions can manipulate as a unit

☐ FORM—a collection of one or more pages of fields

☐ FIELDTYPEa field attribute that determines what kind of data may occupy the field

■ Every ETI function is introduced with a SYNOPSIS that describes the type of its arguments and return value, if any. The first line of the SYNOPSIS proper describes the routine, while the following lines describe its arguments. On each line, the type of the return value or arguments precedes their names. As an example, consider

SYNOPSIS

int set_menu_win (menu, window)
MENU * menu;
WINDOW * window;

This says that the function **set_menu_win()** returns a value of type **int** and that it takes two arguments, **menu** and **window**. The argument **menu** is of type MENU * (pointer to a menu), while the argument **window** is of type WINDOW * (pointer to a window).

■ The terms *window, panel, menu,* and *form* are often shorthand for the phrases *window pointer, panel pointer, menu pointer,* and *form pointer,* respectively. All ETI routines pass or return pointers to these objects, not the objects themselves.

What is ETI?

ETI is a set of C library routines that promote the development of application programs that display and manipulate windows, panels, menus, and forms and run under the UNIX System. The rest of this section explains the nature of these libraries and the connection between ETI and the **terminfo** library and database.

The ETI Libraries

ETI consists of the following libraries.

■ low-level (**curses**)

■ **panel**

■ **menu**

■ **form**

■ TAM Transition.

The routines are C functions and macros; many of them resemble routines in the standard C library. For example, there's a routine **printw()** that behaves much like **printf**(3S) and another routine **getch()** that behaves like **getc**(3S). The automatic teller program at your bank might use **printw()** to print its menus and **getch()** to accept your requests for withdrawals (or, better yet, deposits). A visual screen editor like the UNIX System screen editor **vi**(1) (see the *User's/System Administrator's Reference Manual*) might also use these and other ETI routines.

A major feature of ETI is cursor optimization. Cursor optimization minimizes the amount a cursor has to move around a screen to update it. For example, if you designed a screen editor program with ETI routines and edited the sentence

        ETI is a great package for creating forms and menus.

to read

        ETI is the best package for creating forms and menus.

the program would change only the best in place of a great. The other characters would be preserved. Because the amount of data transmitted—the output—is minimized, cursor optimization is also referred to as output

optimization.

Cursor optimization takes care of updating the screen in a manner appropriate for the terminal ón which an ETI program is run. This means that ETI can do whatever is required to update many different terminal types. It searches the **terminfo** database (described below) to find the correct description for a terminal.

How does cursor optimization help you and those who use your programs? First, it saves you time in describing in a program how you want to update screens. Second, it saves a user's time when the screen is updated. Third, it reduces the load on your UNIX System's communication lines when the updating takes place. Fourth, you don't have to worry about the myriad of terminals on which your program might be run.

Here's a simple ETI program. It uses some of the basic ETI routines to move a cursor to the middle of a terminal screen and print the character string BullsEye. Each of these routines is described later in this section. For now, just look at their names, and you will get an idea of what each of them does:

```
#include <curses.h>

main()
{
    initscr();

    move( LINES/2 - 1, COLS/2 - 4 );
    addstr("Bulls");
    refresh();
    addstr("Eye");
    refresh();
    endwin();
}
```

Figure 10-1: A Simple ETI Program

The ETI/terminfo Connection

terminfo is both a set of routines that make use of the capabilities of a wide range of terminals and a database that contains descriptions of the terminals that can be used with ETI. Its use as a database is our concern here. See the section "Working with **terminfo** Routines" for details on its use as a set of routines.

A screen management program with ETI routines refers to the **terminfo** database at run time to obtain the information it needs about the terminal being used—what we'll call the current terminal from here on.

Suppose, for instance, that you are using an AT&T Teletype 5425 terminal to run the simple ETI program shown in Figure 10-1. To execute properly, the program needs to know how many lines and columns the terminal screen has to print the BullsEye in the middle of it. The description of the AT&T Teletype 5425 in the **terminfo** database has this information, as well as other information about the terminal's capabilities and how it performs various operations — for example, how its control characters are interpreted. All ETI needs to know before it goes looking for the information is the name of your terminal.

You tell the program the name by putting it in the environment variable **$TERM** when you log in or by setting and exporting **$TERM** in your **.profile** file [see **profile**(4)]. Knowing **$TERM**, an ETI program run on the current terminal can search the **terminfo** database to find the correct terminal description.

For example, assume that the following lines are in a **.profile**:

```
TERM=5425
export TERM
tput init
```

The first line names the terminal type, and the second line exports it. (See **profile**(4) in the *Programmer's Reference Manual*.) The third line of the example tells the UNIX System to initialize the current terminal. That is, it makes sure that the terminal is set up according to its description in the **terminfo** database. (The order of these lines is important. **$TERM** must be defined and exported first, so that when **tput**(1) [see the *User's/System Administrator's Reference Manual*] is called the proper initialization for the current terminal will take place.) If you had these lines in your **.profile** and you ran an ETI

program, the program would get the information that it needs about your terminal from the file **/usr/lib/terminfo/5/5425** in the database, which provides a match for **$TERM**. For more information about the **terminfo** database, see the section "Working with **terminfo** Routines".

Basic ETI Programming

This section describes the low-level routines and other components that every ETI program needs to work properly. It tells you how to compile and run ETI applications using the low-level libraries and introduces important concepts (such as refreshing) that recur throughout this document.

What Every ETI Program Needs

All ETI programs need to include the header files **<menu.h>**, **<form.h>**, and **<panel.h>** and call the routines **initscr()**, **refresh()** or similar routines, and **endwin()**. Some of the other header files, however, include file **curses.h**.

The Header File <curses.h>

The header files **<menu.h>**, **<form.h>**, and **<panel.h>** define several global variables and data structures.

To begin, let's consider the variables and data structures defined. **<curses.h>**, among other things, defines the integer variables **LINES** and **COLS**; when an ETI program is run on a particular terminal, these variables are assigned the vertical and horizontal dimensions of the terminal screen, respectively, by the routine **initscr()** described below.

The integer variables **COLORS** and **COLOR_PAIRS** are also defined in **<curses.h>**. These will be assigned, respectively, the maximum number of colors and color-pairs the terminal can support. These variables are initialized by the **start_color()** routine. (See the section "Color Manipulation.")

> **NOTE**
>
> **LINES** and **COLS** are external (global) variables that represent the size of a terminal screen. Two similar variables, **$LINES** and **$COLUMNS**, may be set in a user's shell environment; an ETI program uses the environment variables to determine the size of a screen. Whenever we refer to the environment variables in this section, we will use the **$** to distinguish them from the C declarations in the **<curses.h>** header file.
>
> For more information about these variables, see the following sections: "The Routines **initscr()**, **refresh()**, and **endwin()**" and "More about **initscr()** and Lines and Columns."

The header files define the integer constants OK, E_OK, ERR, and others listed in the following sections. ETI routines that return **int** values return these constants under the following conditions:

OK   returned if a low-level or panel function completes properly

E_OK   returned if a menu or form function does so

ERR   returned if a low-level or panel function encounters an error

The other error values returned by the high-level functions are described in the appropriate sections below.

Now let's consider the macro definitions. **<curses.h>** defines many ETI routines as macros that call other macros or ETI routines. For instance, the simple routine **refresh()** is a macro. The line

```
#define refresh() wrefresh(stdscr)
```

shows that when **refresh** is called, it is expanded to call the ETI routine **wrefresh()**. In turn, **wrefresh()** (although it is not a macro) calls the two ETI routines **wnoutrefresh()** and **doupdate()**. Many other routines also group two or three routines together to achieve a particular result.

CAUTION
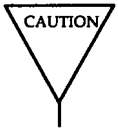
Macro expansion in ETI programs may cause problems with certain sophisticated C features, such as the use of automatic incrementing variables.

One final point about **<curses.h>**: it automatically includes **<stdio.h>** and the **<termio.h>** tty driver interface file. Including either file again in a program is harmless but wasteful.

The Routines initscr(), refresh(), endwin()

The routines **initscr()**, **refresh()**, and **endwin()** initialize a terminal screen to an "in ETI state," update the contents of the screen, and restore the terminal to an "out of ETI state," respectively. Consider the simple program introduced earlier and reproduced in Figure 10-2.

```
#include <curses.h>

main()
{
    initscr();      /* initialize terminal settings and <curses.h>
                       data structures and variables */

    move( LINES/2 - 1, COLS/2 - 4 );
    addstr("Bulls");
    refresh();      /* send output to (update) terminal screen */
    addstr("Eye");
    refresh();      /* send more output to terminal screen */
    endwin();       /* restore all terminal settings */
}
```

Figure 10-2: The Purposes of **initscr()**, **refresh()**, and **endwin()** in a Program

An ETI program usually starts by calling **initscr()**; your program should call **initscr()** only once. This routine uses the environment variable **$TERM** to determine what terminal is being used. (See the Chapter 1 section, "The ETI/**terminfo** Connection," for details.) It then initializes all the declared data structures and other variables from <**curses.h**>. For example, **initscr()** would initialize **LINES** and **COLS** for the sample program on whatever terminal it was run. If the TELETYPE 5425 terminal were used, this routine would initialize **LINES** to 24 and **COLS** to 80. Finally, this routine writes error messages to **stderr** and exits if errors occur.

During the execution of the program, output and input is handled by routines like **move()** and **addstr()** in the sample program. For example,

        move(LINES/2 - 1, COLS/2 - 4);

says to move the cursor to the left of the middle of the screen. The line

        addstr("Bulls");

says to write the character string Bulls. For example, if the TELETYPE 5425 terminal were used, these routines would position the cursor and write the

character string at (11,36).

> **NOTE**
> All ETI routines that move the cursor move it from its home position in the upper left corner of a screen. The **(LINES,COLS)** coordinate at this position is (0,0) not (1,1). Notice that the vertical coordinate is given first and the horizontal second, which is the opposite of the common 'x,y' order of screen (or graph) coordinates. The -1 in the sample program takes the (0,0) position into account to place the cursor on the center line of the terminal screen.

Routines like **move()** and **addstr()** do not actually change a physical terminal screen when they are called. The screen is updated only when **refresh()** is called after one or more windows (internal representations of the screen) are updated. This is a very important concept, which we discuss below under "More about **refresh()** and Windows."

Finally, an ETI program ends by calling **endwin()**. This routine restores all terminal settings and positions the cursor at the lower left corner of the screen.

Compiling an ETI Program

You compile programs that include ETI routines as C language programs. This means that you use the **cc(1)** command (documented in the *Programmer's Reference Manual*) to invoke the C compiler. (See Chapter 2 for details.)

The routines are usually stored in the library **/usr/lib/libX.a**, where X signifies either **curses**, **panel**, **menu**, or **form**, depending on which library your program needs. To direct the link editor to search this library, you must use the **-l** option with the **cc** command.

The general command line for compiling an ETI program follows:

    cc *file*.**c** *[-lX]* **-lcurses** **-o** *file*

where **X** is either **panel**, **menu**, or **form**; *file*.c is the name of the source program; and *file* is the executable object module. See the appropriate section below for more information.

Using the TAM Transition Library

Some users may have applications using the TAM library routines that originally ran on the UNIX System PC. "The TAM Transition Library," Appendix B of this document, explains how to compile and run these

applications.

Running an ETI Program

ETI programs count on certain information being in a user's environment to run properly. Specifically, users of a program should usually include the following three lines in their **.profile** files:

```
TERM=current terminal type
export TERM
tput init
```

For an explanation of these lines, turn again to the section "The ETI/**terminfo** Connection" in this chapter. Users of an ETI program could also define the environment variables $LINES, $COLUMNS, and $TER-MINFO in their **.profile** files. However, unlike $TERM, these variables do not have to be defined.

If an ETI program does not run as expected, you might want to debug it with **sdb**(1), which is documented in the *Programmer's Reference Manual*. When using **sdb**, you have to keep a few points in mind. First, an ETI program is interactive and always has knowledge of where the cursor is located. An interactive debugger like **sdb**, however, may cause changes to the contents of the screen of which the ETI program is not aware.

Second, an ETI program doesn't output to a window until **refresh**() or a similar routine is called. Because output from the program may be delayed, debugging the output for consistency may be difficult.

Third, setting break points on ETI routines that are macros, such as **refresh**(), does not work. You have to use the routines defined for these macros, instead; for example, you have to use **wrefresh**() instead of **refresh**(). See the above section, "The Header File <**curses.h**>," for more information about macros.

More about initscr() and Lines and Columns

After determining a terminal's screen dimensions, **initscr**() sets the variables **LINES** and **COLS**. These variables are set from the **terminfo** variables **lines** and **columns**. These, in turn, are set from the values in the **terminfo** database, unless these values are overridden by the values of the environment $LINES and $COLUMNS.

More about refresh() and Windows

As mentioned above, ETI routines do not update a terminal until **refresh()** is called. Instead, they write to an internal representation of the screen called a window. When **refresh()** is called, all the accumulated output is sent from the window to the current terminal screen.

A window acts a lot like a buffer does when you use a UNIX System editor. When you invoke **vi**(1) (see the *User's/System Administrator's Reference Manual*), for instance, to edit a file, the changes you make to the contents of the file are reflected in the buffer. The changes become part of the permanent file only when you use the **w** or **ZZ** command. Similarly, when you invoke a screen program made up of ETI routines, they change the contents of a window. The changes become part of the current terminal screen only when **refresh()** is called.

<curses.h> supplies a default window named **stdscr** (standard screen), which is the size of the current terminal's screen, for all programs using ETI routines. The header file defines **stdscr** to be of the type **WINDOW***, a pointer to a C structure which you might think of as a two-dimensional array of characters representing a terminal screen. The program always keeps track of what is on the physical screen, as well as what is in **stdscr**. When **refresh()** is called, it compares the two screen images and sends a stream of characters to the terminal that make the physical screen look like **stdscr**. An ETI program considers many different ways to do this, taking into account the various capabilities of the terminal and similarities between what is on the screen and what is on the window (**stdscr**). It optimizes output by printing as few characters as possible. Figure 10-3 illustrates what happens when you execute the sample ETI program that prints **BullsEye** at the center of a terminal screen. Notice in the figure that the terminal screen retains whatever garbage is on it until the first **refresh()** is called. This **refresh()** clears the screen and updates it with the current contents of **stdscr**.
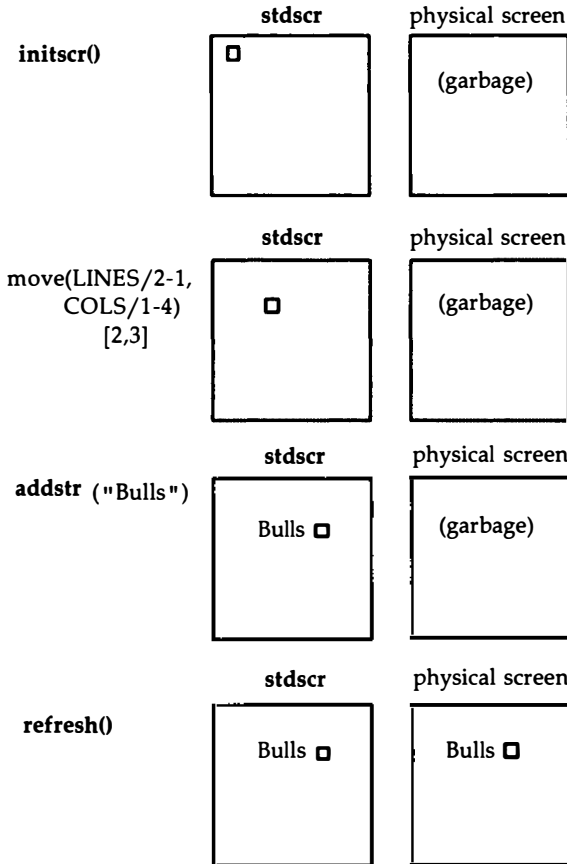
	stdscr	physical screen
initscr()	▯	(garbage)
move(LINES/2-1, **COLS/1-4)** **[2,3]**	▯	(garbage)
addstr ("Bulls")	Bulls ▯	(garbage)
refresh()	Bulls ▯	Bulls ▯

Figure 10-3: The Relationship between **stdscr** and a Terminal Screen

```
                    stdscr              physical screen

addstr ("Eye")  ┌──────────┐          ┌──────────┐
                │          │          │          │
                │ BullsEye □│          │  Bulls □ │
                │          │          │          │
                └──────────┘          └──────────┘

                    stdscr              physical screen

refresh()       ┌──────────┐          ┌──────────┐
                │          │          │          │
                │ BullsEye □│          │ BullsEye □│
                │          │          │          │
                └──────────┘          └──────────┘

                    stdscr              physical screen

endwin()        ┌──────────┐          ┌──────────┐
                │          │          │  BullsEye │
                │ BullsEye □│          │          │
                │          │          │ □        │
                └──────────┘          └──────────┘
```
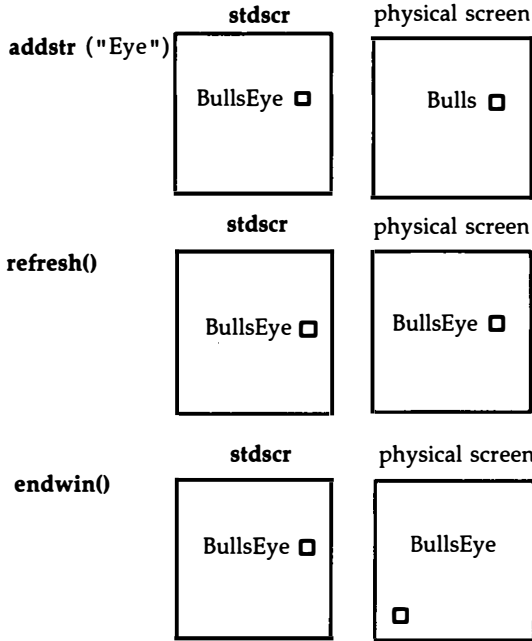
Figure 10-3: The Relationship Between **stdscr** and a Terminal Screen (**continued**)

———

You can create other windows and use them instead of **stdscr**. Windows are useful for maintaining several different screen images. For example, many data entry and retrieval applications use two windows: one to control input and output and one to print error messages that don't mess up the other window. It's possible to subdivide a screen into many windows, refreshing each one of them as desired. And it's possible to create a window within a window; the smaller window is called a subwindow. See the section, "Windows," for more information.

Pads

Some ETI routines are designed to work with a special type of window called a pad. A pad is a window whose size is not restricted by the size of a screen or associated with a particular part of a screen. You can use a pad when you have a particularly large window or only need part of the window on the screen at any one time. For example, you might use a pad for an application with a spread sheet.

Figure 10-4 represents what a pad, a subwindow, and some other windows could look like in comparison to a physical screen.
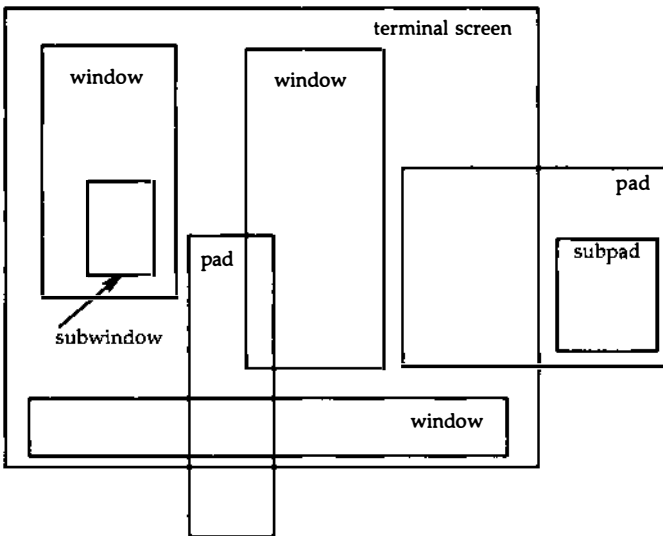
Figure 10-4: Multiple Windows and Pads Mapped to a Physical Screen

The later section "Windows" describes the routines you use to create and use windows and pads. If you'd like to see an ETI program with windows now, turn to the **window** program under the section "ETI Program Examples" in this chapter.

Simple Input and Output

This section explains the numerous functions that enable you to do I/O under the ETI environment. It also covers the set of video attributes and options which can enhance ETI output with striking visual effects.

Output

The routines that low-level ETI provides for writing to **stdscr** are similar to those provided by the **stdio**(3S) library for writing to a file. They let you

- write a character at a time — **addch()**

- write a string — **addstr()**

- format a string from a variety of input arguments — **printw()**

- move a cursor or move a cursor and print character(s) — **move()**, **mvaddch()**, **mvaddstr()**, **mvprintw()**

- clear a screen or a part of it — **clear()**, **erase()**, **clrtoeol()**, **clrtobot()**

Following are descriptions and examples of these routines.
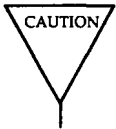
CAUTION / The ETI library provides its own set of input and output functions. You should not use other I/O routines or system calls, like **printf**(3S) and **scanf**(3S), in an ETI program. They may cause undesirable results when you run the program.

addch()

SYNOPSIS

#include <curses.h>

int addch(ch)
chtype ch;

NOTES

- **addch**() writes a single character to **stdscr** and advances the cursor to the next character position.

- The character is of the type **chtype**, which is defined in **<curses.h>**. **chtype** contains data and attributes (see "Output Attributes" in this chapter for information about attributes).

- When working with variables of this type, make sure you declare them as **chtype** and not as the basic type (for example, **unsigned long**) that **chtype** is declared to be in **<curses.h>**. This will ensure future compatibility.

- **addch**() does some translations. For example, it converts

  □ the **<NL>** character to a clear to end of line and a move to the next line

  □ the tab character to an appropriate number of blanks

  □ other control characters to their ˆX notation

- **addch**() normally returns **OK**. The only time **addch**() returns **ERR** is after adding a character to the lower right-hand corner of a window that does not scroll.

- **addch**() is a macro.

EXAMPLE

```
#include <curses.h>

main()
{
    initscr();
    addch('a');
    refresh();
    endwin();
}
```

The output from this program will appear as follows, with 'a' in position 0, 0:

```
 _____
/                                             \
|                                             |
|   a                                         |
|                                             |
|                                             |
|   $□                                        |
|                                             |
_____/
```
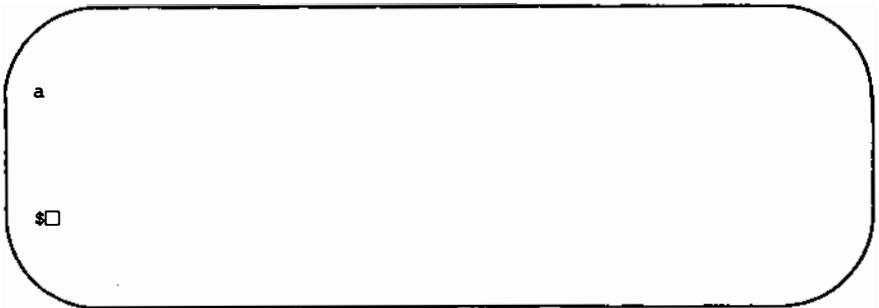
See also the **show** program under "ETI Example Programs" in this chapter.

addstr()

SYNOPSIS

#include <curses.h>

int addstr(str)
char *str;

NOTES

- **addstr()** writes a string of characters to **stdscr**.

- **addstr()** calls **addch()** to write each character.

- **addstr()** follows the same translation rules as **addch()**.

- **addstr()** returns **OK** on success and **ERR** on error.

- **addstr()** is a macro.

EXAMPLE

Recall the sample program that prints the character string **BullsEye**. See Figures 10-2, 10-3, and 10-4.

printw()

SYNOPSIS

#include <curses.h>

int printw(fmt [,arg...])
char *fmt

NOTES

- **printw()** handles formatted printing on **stdscr**.

- Like **printf**, **printw()** takes a format string and a variable number of arguments.

- Like **addstr()**, **printw()** calls **addch()** to write the string.

- **printw()** returns **OK** on success and **ERR** on error.

EXAMPLE

```
#include <curses.h>

main()
{
    char* title = "Not specified";
    int no = 0;

        /* Missing code. */

    initscr();

        /* Missing code. */

    printw("%s is not in stock.\n", title);
    printw("Please ask the cashier to order %d for you.\n", no);

    refresh();
    endwin();
}
```
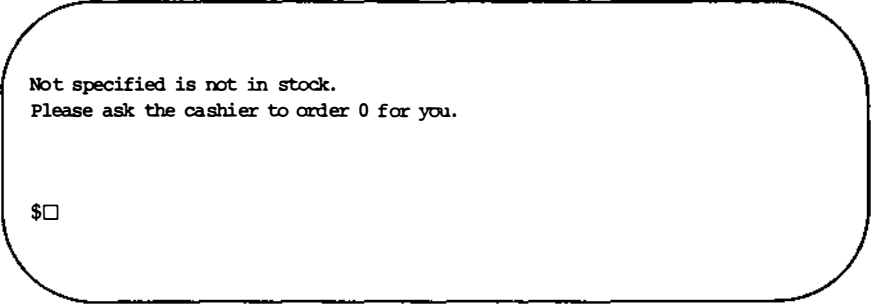
The output from this program will appear as follows:

```
Not specified is not in stock.
Please ask the cashier to order 0 for you.


$□
```

move()

SYNOPSIS

#include <curses.h>

int move(y, x);
int y, x;

NOTES
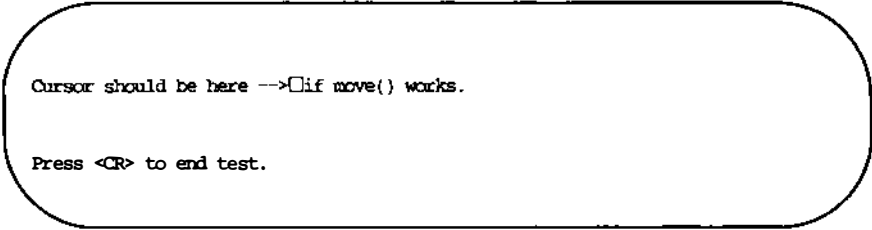
- **move()** positions the cursor for **stdscr** at the given row **y** and the given column **x**.

- Notice that **move()** takes the **y** coordinate before the **x** coordinate. The upper left-hand coordinates for **stdscr** are (0,0), the lower right-hand (**LINES** - 1, **COLS** - 1). See the section "The Routines **initscr()**, **refresh()**, and **endwin()**" for more information.

- **move()** may be combined with the write functions to form

  □ **mvaddch(y, x, ch)**, which moves to a given position and prints a character

  □ **mvaddstr(y, x, str)**, which moves to a given position and prints a string of characters

  □ **mvprintw(y, x, fmt** [,*arg*...]),
     which moves to a given position and prints a formatted string.

- **move()** returns **OK** on success and **ERR** on error. Trying to move to a screen position of less than (0,0) or more than (**LINES** - 1, **COLS** - 1) causes an error.

- **move()** is a macro.

EXAMPLE

```
#include <curses.h>

main()
{
    initscr();
    addstr("Cursor should be here --> if move() works.");
    printw("\n\n\nPress <CR> to end test.");
    move(0,25);
    refresh();
    getch();      /* Gets <CR>; discussed below. */
    endwin();
}
```
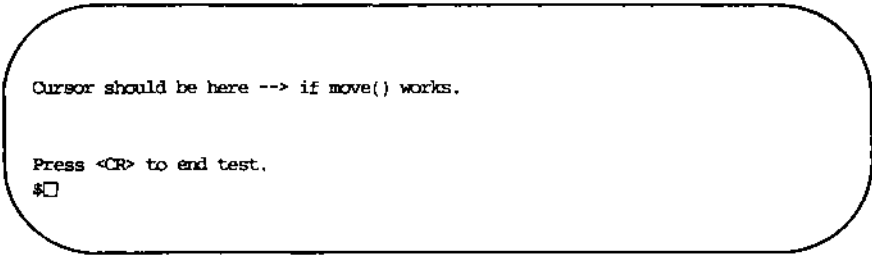
Here's the output generated by running this program:

```
    Cursor should be here -->☐if move() works.


    Press <CR> to end test.
```

After you press **<CR>**, the screen looks like this:

```
    Cursor should be here --> if move() works.


    Press <CR> to end test.
    $☐
```

See the **scatter** program under "ETI Program Examples" in this chapter for another example using **move()**.

clear() **and** erase()

SYNOPSIS

#include <curses.h>

int clear()
int erase()

NOTES

- Both routines change **stdscr** to all blanks.

- **clear()** assumes that the screen may have garbage that it doesn't know about; this routine first calls **erase()** and then **clearok()** which clears the physical screen completely on the next call to **refresh()** for **stdscr**. See the low-level ETI or **curses**(3X) manual page for more information about **clearok()**.

- **initscr()** automatically calls **clear()**.

- **clear()** and **erase()** always return OK.

- Both routines are macros.

clrtoeol() and clrtobot()

SYNOPSIS

#include <curses.h>

int clrtoeol()
int clrtobot()

NOTES

■ **clrtoeol**() changes the remainder of a line to all blanks.

■ **clrtobot**() changes the remainder of a screen to all blanks.

■ Both begin at the current cursor position inclusive.

■ Neither returns any useful value.

EXAMPLE

The following sample program uses **clrtobot**().

```
#include <curses.h>

main()
{
    initscr();
    addstr("Press <CR> to delete from here to the end of the line and on.");
    addstr("\nDelete this too.\nAnd this.");
    move(0,30);
    refresh();
    getch();
    clrtobot();
    refresh();
    endwin();
}
```

Here's the output generated by running this program:

```
Press <CR> to delete from here□to the end of the line and on.
Delete this too.
And this.
```

Notice the two calls to **refresh**(): one to send the full screen of text to a terminal, the other to clear from the position indicated to the bottom of a screen.

Here's what the screen looks like when you press **<CR>**:

```
Press <CR> to delete from here

$□
```

See the **show** and **two** programs under "Program Examples" for other uses of **clrtoeol**().

Input

Low-level routines for reading from the current terminal are similar to those provided by the **stdio**(3S) library for reading from a file. They let you do the following:

- read one character at a time — **getch()**

- read a <**NL**>-terminated string — **getstr()**

- parse input, converting and assigning selected data to an argument list — **scanw()**

The primary routine is **getch()**, which processes a single input character and then returns that character. This routine is like the C library routine **getchar()**(3S) except that it makes several terminal- or system-dependent options available that are not possible with **getchar()**. For example, you can use **getch()** with the ETI routine **keypad()**, which allows a low-level ETI program to interpret extra keys on a user's terminal, such as arrow keys, function keys, and other special keys that transmit escape sequences, and treat them as just another key. See the descriptions of **getch()** and **keypad()** on the **curses**(3X) manual page for more information about **keypad()**.

The following pages describe and give examples of the basic routines for getting input in a screen program.

getch()

SYNOPSIS

#include <curses.h>

int getch()

NOTES

- **getch**() reads a single character from the current terminal.

- **getch**() returns the value of the character or **ERR** on 'end of file,' receipt of signals, or nonblocking read with no input.

- **getch**() is a macro.

- See the discussions about **echo(), noecho(), cbreak(), nocbreak(), raw(), noraw(), halfdelay(), nodelay(),** and **keypad**() on the following pages and in **curses**(3X).
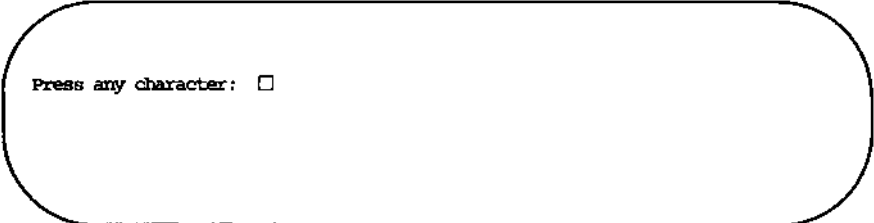
EXAMPLE

```
#include <curses.h>

main()
{
    int ch;

    initscr();
    cbreak();          /* Explained later in the section "Input Options" */
    addstr("Press any character:  ");
    refresh();
    ch = getch();
    printw("\n\n\nThe character entered was a '%c'.\n", ch);
    refresh();
    endwin();
}
```

The output from this program follows. The first **refresh()** sends the **addstr()** character string from **stdscr** to the terminal.

Press any character: ☐

Now assume that a **w** is typed at the keyboard. **getch()** accepts the character and assigns it to **ch**. Finally, the second **refresh()** is called and the screen appears as follows:

```
Press any character:  w


The character entered was a 'w'.


$□
```

For another example of **getch()**, see the **show** program under "Program Examples" in this chapter.

getstr()

SYNOPSIS

#include <curses.h>

int getstr(str)
char *str;

NOTES

■ **getstr()** reads characters and stores them in a buffer until a **<CR>**,
<NL>, or **<ENTER>** is received from **stdscr**. **getstr()** does not check
for buffer overflow.

■ The characters read and stored are in a character string.

■ **getstr()** is a macro; it calls **getch()** to read each character.

■ **getstr()** returns **ERR** if **getch()** returns **ERR** to it. Otherwise it returns
OK.

■ See the discussions on **echo()**, **noecho()**, **cbreak()**, **nocbreak()**, **raw()**,
noraw(), **halfdelay()**, **nodelay()**, and **keypad()** on the following pages
and in ETI **curses**(3X).

EXAMPLE

```
#include <curses.h>

main()
{
char str[256];

    initscr();
    cbreak();        /* Explained later in the section "Input Options" */
    addstr("Enter a character string terminated by <CR>:\n\n");
    refresh()
    getstr(str);
    printw("\n\n\nThe string entered was \n'%s'\n", str);
    refresh();
    endwin();
}
```

Assume you entered the string 'I enjoy learning about the UNIX System.' The final screen (after entering <CR>) would appear as follows:

```
Enter a character string terminated by <CR>:

I enjoy learning about the UNIX System.


The string entered was
'I enjoy learning about the UNIX System.'

$□
```

scanw()

SYNOPSIS

#include <curses.h>

int scanw(fmt [, arg...])
char *fmt;

NOTES

- **scanw()** calls **getstr()** and parses an input line.

- Like **scanf**(3S), **scanw()** uses a format string to convert and assign to a variable number of arguments.

- **scanw()** returns the same values as **scanf()**.

- See **scanf**(3S) for more information.
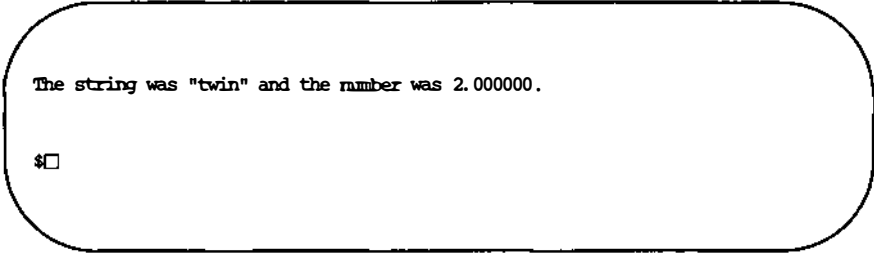
EXAMPLE

```
#include <curses.h>

main()
{
    char string[100];
    float number;

    initscr();
    cbreak();               /* Explained later in the  */
    echo();                 /* section "Input Options" */
    addstr("Enter a number and a string separated by a comma: ");
    refresh();
    scanw("%f,%s",&number,string);
    clear();
    printw("The string was \"%s\" and the number was %f.",string,number);
    refresh();
    endwin();
}
```

Notice the two calls to **refresh()**. The first call updates the screen with the character string passed to **addstr()**, the second with the string returned from **scanw()**. Also notice the call to **clear()**. Assume you entered the following when prompted: **2,twin**. After running this program, your terminal screen would appear, as follows:

The string was "twin" and the number was 2.000000.

$□

Output Attributes

When we talked about **addch**(), we said that it writes a single character of the type **chtype** to **stdscr**. **chtype** has two parts: a part with information about the character itself and another part with information about a set of attributes associated with the character. The attributes allow a character to be printed in reverse video, bold, a particular color, underlined, and so on.

stdscr always has a set of current attributes that it associates with each character as it is written. However, using the routine **attrset**() and related ETI routines described below, you can change the current attributes. Below is a list of the attributes and what they mean:

- A_BLINK—blinking
- A_BOLD—extra bright or bold
- A_DIM—half bright
- A_REVERSE—reverse video
- A_STANDOUT—a terminal's best highlighting mode
- A_UNDERLINE—underlining
- A_ALTCHARSET—alternate character set (see the section "Drawing Lines and Other Graphics" in this chapter)
- COLOR_PAIR(n)—change foreground and background colors (see the section on "Color Manipulation" in this section)

To use these attributes, you must pass them as arguments to **attrset**() and related routines; they can also be ORed with the bitwise OR (|) to **addch**().

| NOTE | Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, an ETI program attempts to find a substitute attribute. If none is possible, the attribute is ignored. |

Let's consider a use of one of these attributes. To display a word in bold, you would use the following code:

```
   ...
printw("A word in ");
attrset(A_BOLD);
printw("boldface");
attrset(0);
printw(" really stands out.\n");
   ...
refresh();
```

Attributes can be turned on singly, such as **attrset(A_BOLD)** in the example, or in combination. To turn on blinking bold text, for example, you would use **attrset(A_BLINK| A_BOLD)**. Individual attributes can be turned on and off with the ETI routines **attron()** and **attroff()** without affecting other attributes. **attrset(0)** turns all attributes off, including changes you may have made to foreground and background color.

Notice the attribute called A_STANDOUT. You might use it to make text attract the attention of a user. The particular hardware attribute used for standout is the most visually pleasing attribute a terminal has. Standout is typically implemented as reverse video or bold. Many programs don't really need a specific attribute, such as bold or reverse video, but instead just need to highlight some text. For such applications, the A_STANDOUT attribute is recommended. Two convenient functions, **standout()** and **standend()** can be used to turn on and off this attribute. **standend()**, in fact, turns off all attributes.

In addition to the attributes listed above, there are two bit masks called A_CHARTEXT and A_ATTRIBUTES. You can use these bit masks with the ETI function **inch()** and the C logical AND (**&**) operator to extract the character or attributes of a position on a terminal screen. See the discussion of **inch()** on the **curses**(3X) manual page. A third bit mask, A_COLOR, can be used to extract information about the color-pair field of a position on a terminal screen.

Following are descriptions of **attrset**() and the other ETI routines that you can use to manipulate attributes.

attron(), attrset(), and attroff()

SYNOPSIS

#include <curses.h>

int attron(attrs)
chtype attrs;

int attrset(attrs)
chtype attrs;

int attroff(attrs)
chtype attrs;

NOTES

- **attron**() turns on the requested attribute **attrs** in addition to any that are currently on. **attrs** is of the type **chtype** and is defined in **<curses.h>**.

- **attrset**() turns on the requested attributes **attrs** instead of any that are currently turned on.

- **attroff**() turns off the requested attributes **attrs** if they are on.

- The attributes may be combined using the bitwise OR (I).

- All return **OK**.

EXAMPLE

See the **highlight** program under "Program Examples" in this chapter.

standout() **and** standend()

SYNOPSIS

#include <curses.h>

int standout()
int standend()

NOTES

■ **standout**() turns on the preferred highlighting attribute, A_STANDOUT, for the current terminal. This routine is equivalent to **attron**(A_STANDOUT).

■ **standend**() turns off all attributes. This routine is equivalent to **attrset**(0), where **attrset**() takes the argument **0**.

■ Both always return **OK**.

EXAMPLE

Again, see the **highlight** program under "Program Examples" in this chapter.

Color Manipulation

The ETI color manipulation routines allow you to use colors on an alphanumeric terminal as you would use any other video attribute. You can find out if the ETI library on your system supports the color routines by checking the file **/usr/include/curses.h** to see if it defines the macro **COLOR_PAIR(n)**.

This section begins with a description of the color feature at a general level. Then, the use of color as an attribute is explained. Next, the ways to define color-pairs and change the definitions of colors is explained. Finally, there are guidelines for ensuring the portability of your program, and a section describing the color manipulation routines and macros, with examples.

How the Color Feature Works

Colors are always used in pairs, consisting of a foreground color (used for the character) and a background color (used for the field on which the character is displayed). ETI uses this concept of color-pairs to manipulate colors. In order to use color in a ETI program, you must first define (initialize) the individual colors, then create color-pairs using those colors, and finally, use the color-pairs as attributes.

Actually, the process is even simpler, since ETI maintains a table of initialized colors for you. This table has as many entries as the number of colors your terminal can display at one time. Each entry in the table has three fields: one each for the intensity of the red, green, and blue components in that color.

> NOTE ETI uses RGB (Red, Green, Blue) color notation. This notation allows you to specify directly the intensity of red, green, and blue light to be generated in an additive system. Some terminals use an alternative notation, known as HSL (Hue, Saturation, Luminosity) color notation. Terminals that use HSL can be identified in the **terminfo** database, and ETI will make conversions to RGB notation automatically.

At the beginning of any ETI program that uses color, all entries in the colors table are initialized with eight basic colors, as follows:

	Intensity of Component		
	(R)ed	(G)reen	(B)lue
/* black: 0 */	0	0	0
/* blue: 1 */	0	0	1000
/* green: 2 */	0	1000	0
/* cyan: 3 */	0	1000	1000
/* red: 4 */	1000	0	0
/* magenta: 5 */	1000	0	1000
/* yellow: 6 */	1000	1000	0
/* white: 7 */	1000	1000	1000

The Default Colors Table

Most color alphanumeric terminals can display eight colors at the same time, but if your terminal can display more than eight, then the table will have more than eight entries. The same eight colors will be used to initialize additional entries. If your terminal can display only N colors, where N is less than eight, then only the first N colors shown in the Colors Table will be used.

You can change these color definitions with the routine **init_color()**, if your terminal is capable of redefining colors. If your terminal is not able to change the definition of a color, use of **init_color()** returns **ERR**.

The following color macros are defined in **curses.h** and have numeric values corresponding to their position in the Colors Table.

```
COLOR_BLACK       0
COLOR_BLUE        1
COLOR_GREEN       2
COLOR_CYAN        3
COLOR_RED         4
COLOR_MAGENTA     5
COLOR_YELLOW      6
COLOR_WHITE       7
```

ETI also maintains a table of color-pairs, which has space allocated for as many entries as the number of color-pairs that can be displayed on your terminal screen at the same time. Unlike the colors table, however, there are no default entries in the pairs table: it is your responsibility to initialize any color-pair you want to use with **init_pair()**, before you use it as an attribute.

Each entry in the pairs table has two fields: the foreground color, and the background color. For each color-pair that you initialize, these two fields will each contain a number representing a color in the colors table. (Note that color-pairs can only be made from previously initialized colors.)

The following example pairs table shows that a programmer has used **init_pair()** to initialize color-pair 1 as a blue foreground (entry 1 in the default color table) on yellow background (entry 6 in the default color table). Similarly, the programmer has initialized color-pair 2 as a cyan foreground on a magenta background. Not-initialized entries in the pairs table would actually contain zeros, which corresponds to black on black.

Note that color-pair 0 is reserved for use by ETI and should not be changed or used in application programs.

Color-Pair Number	Foreground	Background
0	0	0
1	1	6
2	3	5
3	0	0
4	0	0
5	0	0
.	.	.
.	.	.
.	.	.

Example of a Pairs Table

Two global variables used by the color routines are defined in <**curses.h**>. They are **COLORS**, which contains the maximum number of colors the terminal supports, and **COLOR_PAIRS**, which contains the maximum number of color-pairs the terminal supports. Both are initialized by the **start_color()** routine to values it gets from the **terminfo** database.

Using the COLOR_PAIR(*n*) **Attribute**

If you choose to use the default color definitions, there are only two things you need to do before you can use the attribute **COLOR_PAIR(*n*)**. First, you must call the routine **start_color()**. Once you've done that, you can initialize color-pairs with the routine **init_pair(***pair, f, b***)**. The first argument, *pair*, is the number of the color-pair to be initialized (or changed), and must be between 1 and **COLOR_PAIRS-1**. The arguments *f* and *b* are the foreground color number and the background color number. The value of these arguments must be between **0** and **COLORS-1**. For example, the two color-pairs in the pairs table described earlier can be initialized in the following way:

```
init_pair (1, COLOR_BLUE, COLOR_YELLOW);
init_pair (2, COLOR_CYAN, COLOR_MAGENTA);
```

Once you've initialized a color-pair, the attribute **COLOR_PAIR(*n*)** can be used as you would use any other attribute. **COLOR_PAIR(*n*)** is a macro, defined in <**curses.h**>. The argument, *n*, is the number of a previously initialized color-pair. For example, you can use the routine **attron()** to turn on a color-pair in addition to any other attributes you may currently have turned on:

```
attron (COLOR_PAIR(1));
```

If you had initialized color-pair 1 in the way shown in the example pairs table, then characters displayed after you turned on color-pair 1 with **attron()** would be displayed as blue characters on a yellow background.

You can also combine **COLOR_PAIR(*n*)** with other attributes, for example,

```
attrset(A_BLINK | COLOR_PAIR(1));
```

would turn on blinking and whatever you have initialized color-pair 1 to be. (**attron()** and **attrset()** are described in the "Controlling Input and Output" section of this chapter, and also on the **curses**(3X) manual page in the *Programmer's Reference Manual*.)

Changing the Definitions of Colors. If your terminal is capable of redefining colors, you can change the predefined colors with the routine **init_color(***color, r, g, b***)**. The first argument, *color*, is the numeric value of the color you want to change, and the last three, *r*, *g*, and *b*, are the intensities of the red, green, and blue components, respectively, that the new color will

contain. Once you change the definition of a color, all occurrences of that color on your screen change immediately.

So, for example, you could change the definition of color 1 (COLOR_BLUE by default), to be light blue, in the following way.

```
init_color (COLOR_BLUE, 0, 700, 1000);
```

Portability Guidelines

Like the rest of ETI the color manipulation routines have been designed to be terminal independent. But it must be remembered that the capability of terminals vary. For example, if you write a program for a terminal that can support 64 color-pairs, that program would not be able to produce the same color effects on a terminal that supports at most 8 color-pairs.

When you are writing a program that may be used on different terminals, you should follow these guidelines:

Use at most seven color-pairs made from at most eight colors.

Programs that follow this guideline will run on most color terminals. Only seven, not eight, color-pairs should be used, even though many terminals support eight color-pairs, because **curses** reserves color-pair 0 for its own use.

Do not use color 0 as a background color.

This is recommended because on some terminals, no matter what color you have defined it to be, color 0 will always be converted to black when used for a background.

Combine color and other video attributes.

Programs that follow this guideline will provide some sort of highlighting, even if the terminal is monochrome. On color terminals, as many of the listed attributes as possible would be used. On monochrome terminals, only the video attributes would be used, and the color attribute would be ignored.

Use the global variables COLORS and COLOR-PAIRS rather than constants when deciding how many colors or color-pairs your program should use.

Other Macros and Routines

There are two other macros defined in <curses.h> that you can use to obtain information from the color-pair field in characters of type **chtype**.

■ **A_COLOR** is a bit mask to extract color-pair information. It can be used to clear the color-pair field, and to determine if any color-pair is being used.

■ **PAIR_NUMBER**(*attrs*) is the reverse of **COLOR_PAIR**(n). It returns the color-pair number associated with the named attribute, *attrs*.

There are two color routines that give you information about the terminal your program is running on. The routine **has_colors()** returns a Boolean value: **TRUE** if the terminal supports colors, **FALSE** otherwise. The routine **can_change_colors()** also returns a Boolean value: **TRUE** if the terminal supports colors *and* can change their definitions, **FALSE** otherwise.

There are two color routines that give you information about the colors and color-pairs that are currently defined on your terminal. The routine **color_content()** gives you a way to find the intensity of the RGB components in an initialized color. It returns **ERR** if the color does not exist or if the terminal cannot change color definitions, **OK** otherwise. The routine **pair_content()** allows you to find out what colors a given color-pair consists of. It returns **ERR** if the color-pair has not been initialized, **OK** otherwise.

These routines are explained in more detail on the **curses(3X)** manual page in the *Programmer's Reference Manual*.

The routines **start_color()**, **init_color()**, and **init_pair()** are described on the following pages, with examples of their use. You can also refer to the program **colors** in the section "Program Examples," at the end of this chapter, for an example of using the attribute of color in windows.

start_color( )

SYNOPSIS

#include <curses.h>

int start_color( )

NOTES

■ This routine must be called if you want to use colors, and before any other color manipulation routine is called. It is good practice to call it right after **initscr( )**.

■ It initializes eight default colors (black, blue, green, cyan, red, magenta, yellow, and white), and the global variables **COLORS** and **COLOR_PAIRS**. If the value corresponding to **COLOR_PAIRS** in the **terminfo** database is greater than 64, **COLOR_PAIRS** will be set to 64.

■ It restores the terminal's colors to the values they had when the terminal was just turned on.

■ It returns **ERR** if the terminal does not support colors, **OK** otherwise.

EXAMPLE

See the example under **init_pair( )**.

init_pair( )

SYNOPSIS

#include <curses.h>

int init_pair (pair, f, b)
short pair, f, b;

NOTES

■ **init_pair( )** changes the definition of a color-pair.

■ Color-pairs must be initialized with **init_pair()** before they can be used as the argument to the attribute macro **COLOR_PAIR(n)**.

■ The value of the first argument, *pair*, is the number of a color-pair, and must be between **1** and **COLOR_PAIRS-1**.

■ The value of the *f* (foreground) and *b* (background) arguments must be between **0** and **COLORS-1**.

■ If the color-pair was previously initialized, the screen will be refreshed and all occurrences of that color-pair will change to the new definition.

■ It returns **OK** if it was able to change the definition of the color-pair, **ERR** otherwise.

EXAMPLE

```
#include <curses.h>

main( )
{
    initscr ( );
    if (start_color ( ) == OK)
    {
        init_pair (1, COLOR_RED, COLOR_GREEN);
        attron (COLOR_PAIR (1));
        addstr ("Red on Green");
        refresh( );
    }
    endwin( );
}
```

Also see the program **colors** in the section "Program Examples."

SYNOPSIS

#include <curses.h>

int init_color(color, r, g, b)
short color, r, g, b;

NOTES

- **init_color()** changes the definition of a color.

- The first argument, *color*, is the number of the color to be changed. The value of *color* must be between **0** and **COLORS-1**.

- The last three arguments, *r, g,* and *b,* are the amounts of red, green, and blue (RGB) components in the new color. The values of these three arguments must be between **0** and **1000**.

- When **init_color()** is used to change the definition of an entry in the colors table, all places where the old color was used on the screen immediately change to the new color.

- It returns **OK** if it was able to change the definition of the color, **ERR** otherwise.

EXAMPLE

```
#include <curses.h>

main( )
{
    initscr( );
    if (start_color == OK)
    {
        init_pair (1, COLOR_RED, COLOR_GREEN);
        attron (COLOR_PAIR (1));
        if (init_color (COLOR_RED, 0, 0, 1000) == OK)
            addstr ("BLUE ON GREEN");
        else
            addstr ("RED ON GREEN");
        refresh ( );
    }
    endwin( );
}
```

Bells, Whistles, and Flashing Lights: beep() and flash()

Occasionally, you may want to get a user's attention. Two low-level ETI routines are designed to help you do this—they let you ring the terminal's chimes and flash its screen.

flash() flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within ear shot of the user. The routine **beep**() can be called when a real beep is desired. (If for some reason the terminal is unable to beep, but able to flash, a call to **beep**() will flash the screen.)

SYNOPSIS

#include <curses.h>

int flash()
int beep()

NOTES

- **flash**() tries to flash the terminal screen, if possible, and, if not, tries to ring the terminal bell.

- **beep**() tries to ring the terminal bell, if possible, and, if not, tries to flash the terminal screen.

- **beep** will not work if you redefine TRUE to something other than 1.

- Neither returns any useful value.

Input Options

The UNIX System does a considerable amount of processing on input before an application ever sees a character. For example, it does the following:

- echoes (prints back) characters to a terminal as they are typed

- interprets an erase character (typically **#**) and a line kill character (typically **@**)

- interprets a CTRL-D (control d) as end of file (EOF)

- interprets interrupt and quit characters

- strips the character's parity bit

- translates <**CR**> to <**NL**>

Because an ETI program maintains total control over the screen, low-level ETI turns off echoing on the UNIX System and does echoing itself. At times, you may not want the UNIX System to process other characters in the standard way in an interactive screen management program. Some ETI routines, **noecho()** and **cbreak()**, for example, have been designed so that you can change the standard character processing. Using these routines in an application controls how input is interpreted. Figure 10-5 shows some of the major routines for controlling input.

Every low-level ETI program accepting input should set some input options. This is because when the program starts running, the terminal on which it runs may be in **cbreak()**, **raw()**, **nocbreak()**, or **noraw()** mode. Although the low-level ETI program starts up in **echo()** mode, as Figure 10-5 shows, none of the other modes are guaranteed.

The combination of **noecho()** and **cbreak()** is most common in interactive screen management programs. Suppose, for instance, that you don't want the characters sent to your application program to be echoed wherever the cursor currently happens to be; instead, you want them echoed at the bottom of the screen. The ETI routine **noecho()** is designed for this purpose. However, when **noecho()** turns off echoing, normal erase and kill processing is still on. Using the routine **cbreak()** causes these characters to be uninterpreted.

| Input | Characters | |
Options	Interpreted	Uninterpreted
Normal 'out of ETI state'	interrupt, quit stripping <CR> to <NL> echoing erase, kill EOF	
Normal ETI 'start up state'	echoing (simulated)	All else undefined.
cbreak() and echo()	interrupt, quit stripping echoing	erase, kill EOF
cbreak() and noecho()	interrupt, quit stripping	echoing erase, kill EOF
nocbreak() and noecho()	break, quit stripping erase, kill EOF	echoing
nocbreak() and echo()	See caution below.	
nl()	<CR> to <NL>	
nonl()		<CR> to <NL>
raw() (instead of cbreak())		break, quit stripping

Figure 10-5: Input Option Settings for ETI Programs
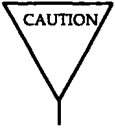
CAUTION Do not use the combination **nocbreak()** and **echo()**. If you use it in a program and also use **getch()**, the program will go in and out of **cbreak()** mode to get each character. Depending on the state of the tty driver when each character is typed, the program may produce undesirable output.

In addition to the routines noted in Figure 10-5, you can use the ETI routines **noraw()**, **halfdelay()**, and **nodelay()** to control input. See the **curses**(3X) manual page for discussions of these routines.

The next few pages describe **noecho()**, **cbreak()** and the related routines **echo()** and **nocbreak()** in more detail.

echo() **and** noecho()

SYNOPSIS

#include <curses.h>

int echo()
int noecho()

NOTES

- **echo()** turns on echoing of characters by ETI as they are read in. This is the initial setting.

- **noecho()** turns off the echoing.

- Neither returns any useful value.

- ETI programs may not run properly if you turn on echoing with **nocbreak()**. See Figure 10-5 and accompanying caution. After you turn echoing off, you can still echo characters with **addch()**.

EXAMPLE

See the **editor** and **show** programs under "Program Examples" in this chapter.

cbreak() **and** nocbreak()

SYNOPSIS

#include < curses.h >
int cbreak()
int nocbreak()

NOTES

- **cbreak()** turns on 'break for each character' processing. A program gets each character as soon as it is typed, but the erase, line kill, and CTRL-D characters are not interpreted.

- **nocbreak()** returns to normal 'line at a time' processing. This is typically the initial setting.

- Neither returns any useful value.

- ETI programs may not run properly if **cbreak()** is turned on and off within the same program or if the combination **nocbreak()** and **echo()** is used.

- See Figure 10-5 and accompanying caution.

EXAMPLE

See the **editor** and **show** programs under "Program Examples" in this chapter.

Windows

An earlier section in this chapter, "More about **refresh**() and Windows," explained what windows and pads are and why you might want to use them. This section describes the ETI routines you use to manipulate and create windows and pads.

Output and Input

The routines that you use to send output to and get input from windows and pads are similar to those you use with **stdscr**. The only difference is that you have to give the name of the window to receive the action. Generally, these functions have names formed by putting the letter **w** at the beginning of the name of a **stdscr** routine and adding the window name as the first parameter. For example, **addch**('c') would become **waddch**(**mywin**, 'c') if you wanted to write the character **c** to the window **mywin**. Here's a list of the window (or **w**) versions of the output routines discussed in "Simple Input and Output."

- **waddch**(*win, ch*)
- **mvwaddch**(*win, y, x, ch*)
- **waddstr**(*win, str*)
- **mvwaddstr**(*win, y, x, str*)
- **wprintw**(*win, fmt [, arg...]*)
- **mvwprintw**(*win, y, x, fmt [, arg...]*)
- **wmove**(*win, y, x*)
- **wclear**(*win*) and **werase**(*win*)
- **wclrtoeol**(*win*) and **wclrtobot**(*win*)
- **wrefresh**(*win*)

You can see from their declarations that these routines differ from the versions that manipulate **stdscr** only in their names and the addition of a *win* argument. Notice that the routines whose names begin with **mvw** take the *win* argument before the *y, x* coordinates, which is contrary to what the names imply. See **curses**(3X) for more information about these routines or the versions of the input routines **getch, getstr**(), and so on that you should use with

windows.

All **w** routines can be used with pads except for **wrefresh()** and **wnoutrefresh()** (see below). In place of these two routines, you have to use **prefresh()** and **pnoutrefresh()** with pads.

The Routines wnoutrefresh() and doupdate()

If you recall from the earlier discussion about **refresh()**, we said that it sends the output from **stdscr** to the terminal screen. We also said that it was a macro that expands to **wrefresh(stdscr)** (see "What Every ETI Program Needs" and "More about **refresh()** and Windows").

The **wrefresh()** routine is used to send the contents of a window (**stdscr** or one that you create) to a screen; it calls the routines **wnoutrefresh()** and **doupdate()**. Similarly, **prefresh()** sends the contents of a pad to a screen by calling **pnoutrefresh()** and **doupdate()**.

Using **wnoutrefresh()**—or **pnoutrefresh()** (this discussion will be limited to the former routine for simplicity)—and **doupdate()**, you can update terminal screens more efficiently than using **wrefresh()** by itself. **wrefresh()** works by first calling **wnoutrefresh()**, which copies the named window to a data structure referred to as the virtual screen. The virtual screen contains what a program intends to display at a terminal. After calling **wnoutrefresh()**, **wrefresh()** then calls **doupdate()**, which compares the virtual screen to the physical screen and does the actual update. If you want to output several windows at once, calling **wrefresh()** will result in alternating calls to **wnoutrefresh()** and **doupdate()**, causing several bursts of output to a screen. However, by calling **wnoutrefresh()** for each window and then **doupdate()** only once, you can minimize the total number of characters transmitted and the processor time used. Figure 10-6 shows a sample program that uses only one **doupdate()**.

```
#include <curses.h>

main()
{
    WINDOW *w1, *w2;

    initscr();
    w1 = newwin(2,6,0,3);
    w2 = newwin(1,4,5,4);
    waddstr(w1, "Bulls");
    wnoutrefresh(w1);
    waddstr(w2, "Eye");
    wnoutrefresh(w2);
    doupdate();
    endwin();
}
```

Figure 10-6: Using **wnoutrefresh()** and **doupdate()**

Notice from the sample that you declare a new window at the beginning of an ETI program. The lines

```
w1 = newwin(2,6,0,3);
w2 = newwin(1,4,5,4);
```

declare two windows named w1 and w2 with the routine **newwin()** according to certain specifications. **newwin()** is discussed in more detail below.

Figure 10-7 illustrates the effect of **wnoutrefresh()** and **doupdate()** on these two windows, the virtual screen, and the physical screen.
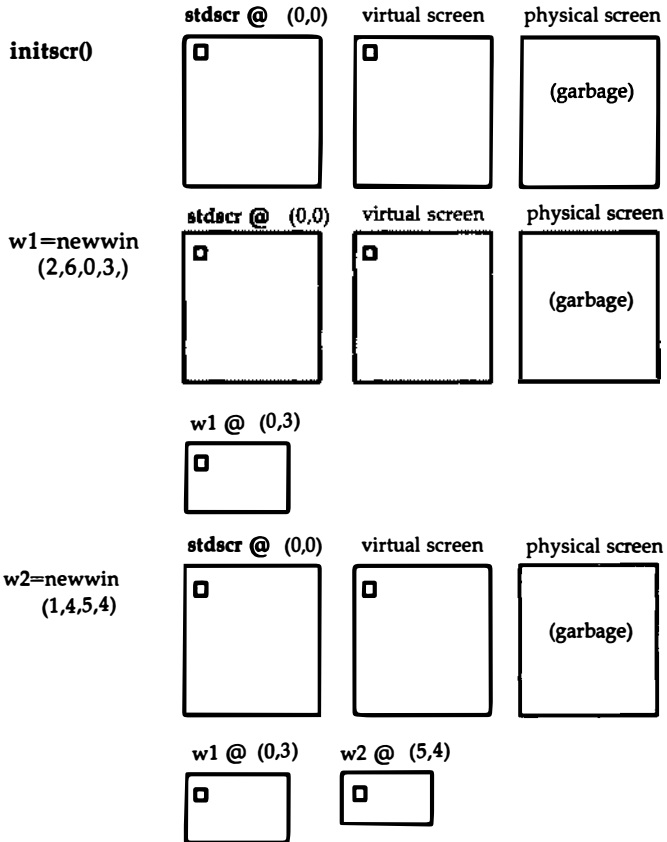
Figure 10-7: The Relationship Between a Window and a Terminal Screen
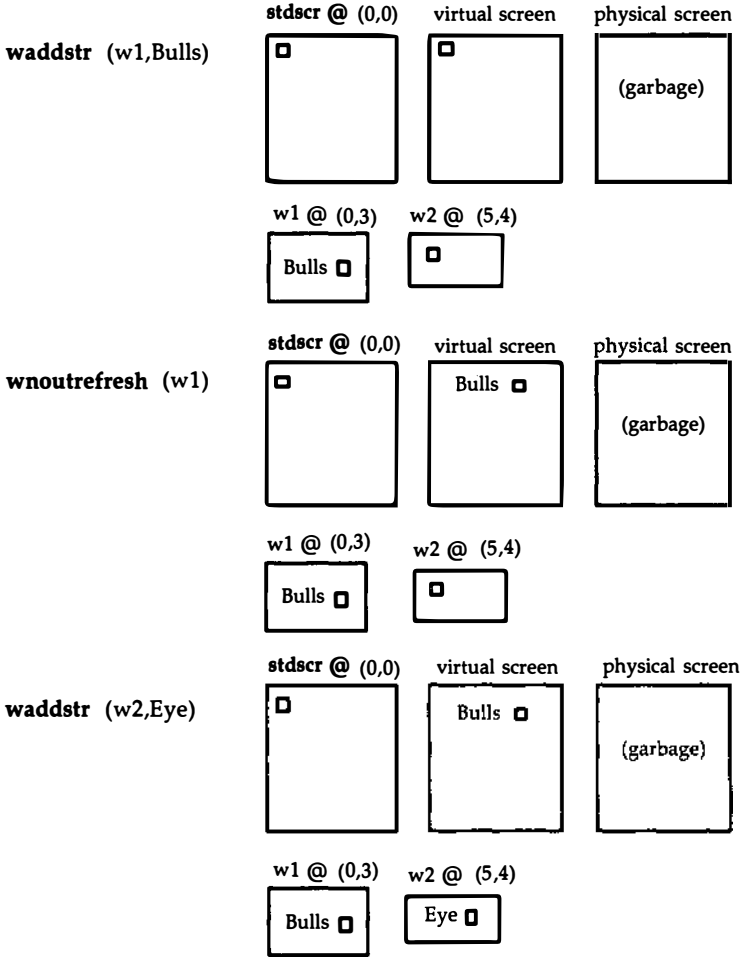
Figure 10-7: The Relationship Between a Window and a Terminal Screen (**continued**)

wnoutrefresh(w2)

stdscr @ (0,0)

virtual screen

physical screen

Bulls

(garbage)

Eye □

w1 @ (0,3)

Bulls □

w2 @ (5,4)

Eye □

doupdate()

stdscr @ (0,0)

virtual screen

physical screen

Bulls

Bulls

Eye □

Eye □

w1 @ (0,3)

Bulls □

w2 @ (5,4)

Eye□

endwin()

stdscr @ (0,0)

virtual screen

physical screen

Bulls

Bulls

Eye □

Eye

w1 @ (0,3)

Bulls □

w2 @ (5,4)

Eye □
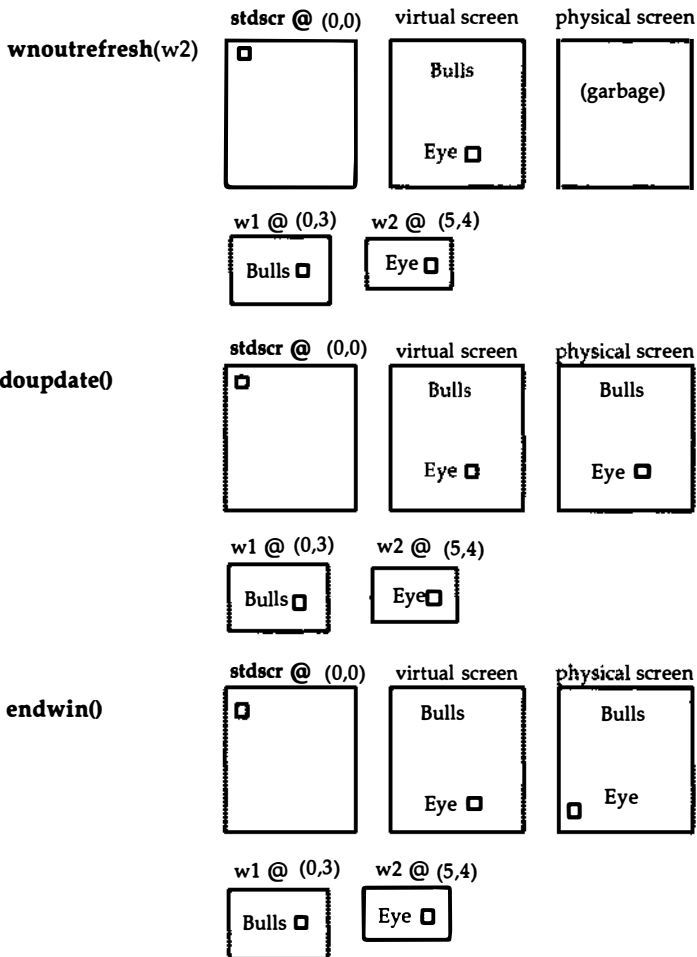
Figure 10-7: The Relationship Between a Window and a Terminal Screen (**continued**)

New Windows

Following are descriptions of the routines **newwin()** and **subwin()**, which you use to create new windows. For information about creating new pads with **newpad()** and **subpad()**, see the **curses**(3X) manual page.

newwin()

SYNOPSIS

#include <curses.h>

WINDOW *newwin(nlines, ncols, begin_y, begin_x)
int nlines, ncols, begin_y, begin_x;

NOTES

■ **newwin()** returns a pointer to a new window with a new data area.

■ The variables **nlines** and **ncols** give the size of the new window.

■ **begin_y** and **begin_x** give the screen coordinates from (0,0) of the upper left corner of the window as it is refreshed to the current screen.

EXAMPLE

Recall the sample program using two windows; see Figure 10-7. Also see the **window** program under "Program Examples" in this chapter.

subwin()

SYNOPSIS

#include <curses.h>

WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
WINDOW *orig;
int nlines, ncols, begin_y, begin_x;

NOTES

- **subwin**() returns a new window that points to a section of another window, **orig**.

- **nlines** and **ncols** give the size of the new window.

- **begin_y** and **begin_x** give the screen coordinates of the upper left corner of the window as it is refreshed to the current screen.

- Subwindows and original windows can accidentally overwrite one another.
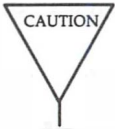
CAUTION Subwindows of subwindows do not work (as of the copyright date of this *Programmer's Guide*).

EXAMPLE

```
#include <curses.h>

main()
{
 WINDOW *sub;

  initscr();
  box(stdscr,'w','w');        /* See the curses(3X) manual page for box() */
  mvwaddstr(stdscr,7,10,"------- this is 10,10");
  mvwaddch(stdscr,8,10,'|');
  mvwaddch(stdscr,9,10,'v');
  sub = subwin(stdscr,10,20,10,10);
  box(sub,'s','s');
  wnoutrefresh(stdscr);
  wrefresh(sub);
  endwin();
}
```

This program prints a border of **w**'s around **stdscr** (the sides of your terminal screen) and a border of **s**'s around the subwindow **sub** when it is run. For another example, see the **window** program under "Program Examples" in this chapter.

ETI Low-Level Interface (curses) to High-Level Functions

In the following sections, we will consider the ETI high-level functions, which create and manipulate panels, menus, and forms. All application programs that use these high-level functions require a set of low-level ETI (**curses**) calls that properly initialize and terminate the programs. For convenience, you may want to isolate these calls in appropriate routines. Figure 10-8 shows one way you might do this. It lists routines to start low-level ETI, terminate it, and handle fatal errors.

```
static char *    PGM     = (char *) 0;    /* program name   */
static int       CURSES  = FALSE; /* is curses initialized ? */

static void start_curses ()/* curses initialization */
{
        CURSES = TRUE;
        initscr ();
        nonl ();
        raw ();
        noecho ();
        wclear (stdscr);
}

static void end_curses ()  /* curses termination */
{
        if (CURSES)
        {
                CURSES = FALSE;
                endwin ();
        }
}

static void error (f, s)    /* fatal error handler */
char * f;
char * s;
{
        end_curses ();
        printf ("%s: ", PGM);
        printf (f, s);
        printf ("0);
        exit (1);
}
```

Figure 10-8: Sample Routines for Low-Level ETI (**curses**) Interface

These house-keeping routines use two global variables, PGM and CURSES.
PGM is initialized with the program's name, while the Boolean CURSES is
initialized with FALSE because **curses** itself has not yet been invoked.

Function **start_curses()** calls the low-level routines previously mentioned and sets CURSES to TRUE to indicate that it has initialized **curses**. Function **end_curses()** checks if **curses** is initialized and, if so, sets the variable CURSES to FALSE and terminates **curses**. The check is necessary because **endwin()** returns an error if called when **curses** is not initialized.

Function **error** is a universal fatal error handler—called whether or not **curses** is initialized. Function **error** first calls **end_curses()** to terminate the program if **curses** is on, and then prints the program's name (PGM) and the associated message. Finally, function **error** terminates the program itself using **exit()**.

Panels

Recall that a window is a rectangular area of the terminal screen on which you can write using the low-level ETI (**curses**) routines. You can create many windows on a screen, but if they overlap, portions of some windows intended to be hidden may nonetheless be visible when you use the low-level routines alone. To solve this problem, ETI uses the notion of a panel—a rectangle of text with depth.

Panels have depth only in relation to other panels and **stdscr**, which lies beneath all panels. The set of currently visible panels comprises the deck of panels.

Compiling and Linking Panel Programs

To use the panel routines, specify

```
#include <panel.h>
```

in your C program files and compile and link with the command line

cc [*flags*] *files* **-lpanel -lcurses** [*libraries*]

Creating Panels

This function creates a new panel on top of all existing panels in the deck. Its argument is a pointer to a window.

```
SYNOPSIS


PANEL *new_panel (window)
WINDOW *window;        /* curses window to be associated with
                               new panel */
```

A pointer to the panel is returned if the panel is created; otherwise, the function returns NULL. The **new_panel()** operation fails if there is insufficient memory or if the window pointer argument is NULL. The window whose address is passed as an argument becomes associated with the panel. The size and location of the panel are the same as that of the low-level ETI **(curses)** window.

To create a panel, create a window, save the pointer to it, and use the pointer as an argument to **new_panel()**.

```
WINDOW *win;
PANEL *pptr;

win = newwin(2,6,0,3);
pptr = new_panel(win);  /* after execution, pptr stores pointer to
                               new panel */
```

Note that the newly created panel does not automatically have any adornments such as titles or borders. If you want your panel to have them, you must call appropriate low-level ETI routines with the panel's window as the argument.

When you create a new panel, it is automatically placed on top of the panel deck. Later, when you call **doupdate()** to adjust the visibility of all panels, the top panel is completely visible. On lower levels, a portion of a panel is visible only when no region of another panel is above it. Where two panels overlap, the higher one hides the lower. (The higher one is the newer one if neither has changed its position in the panel deck because of calls to **top_panel()**, **bottom_panel()**, or **show_panel()** described below.) If the panels do not overlap, the new panel is still logically above the old one. Their relative depth is not apparent until one of them is moved and overlaps the other.

Elementary Panel Window Operations

This section explains how you can fetch pointers to panel windows, change the windows associated with panels, and move panel windows to new locations on the screen.

Fetching Pointers to Panel Windows

Each panel has a low-level ETI window associated with it. To retrieve a pointer to this window, use function **panel_window()**.

```
SYNOPSIS

WINDOW *panel_window(panel)
PANEL *panel;            /* Panel whose window pointer is returned */
```

The function returns NULL if the panel pointer argument is NULL.

In general, you may use this returned pointer as an argument to any standard low-level (**curses**) routine that takes a pointer to a window as an argument. For example, you can insert a character **c** at a location **y,x** in a panel window with the function **mvwinsch(win,y,x,c)**, where **win** is the window pointer returned by **panel_window()**.

```
WINDOW *win;
PANEL *panel;
int y, x;
chtype c;

win = panel_window(panel);
mvwinsch(win,y,x,c);
```

Changing Panel Windows

To replace a panel's pointer to a window with a pointer to another window, call function **replace_panel()**. After the call, the panel remains at the same level within the panel deck.

SYNOPSIS

```
int replace_panel (panel, window)
PANEL *panel;          /* Panel with window to be replaced */
WINDOW *window;             /* New window pointer for panel */
```

This function returns OK if the operation is successful. If not, it returns ERR and leaves the original panel unchanged. Operation **replace_panel()** fails if the window pointer is NULL or there is insufficient memory.

To associate a panel with window **win1** and later replace its window by **win2**, you can write the following:

```
WINDOW *win1, win2;
PANEL *panel;

panel = new_panel(win1);

            /* intervening processing with win1 as panel window */

replace_panel(panel, win2); /* change window associated with panel to win2 */
```

Once you have created additional windows with the low-level function **newwin()**, you in effect can reshape panel windows by using **replace_panel()**. To do so leaves the contents of the two windows unchanged.

Moving Panel Windows on the Screen

You should not move a panel's window by calling the low-level function **mvwin()** directly. To update the screen correctly, the panels subsystem must know the location of all panel windows, but function **mvwin()** does not inform the panels subsystem of the window's new location. To move a panel's window, you must call the function **move_panel()**, which moves a panel and its associated window and informs the panels subsystem of the move.

SYNOPSIS

```
int move_panel (panel, firstrow, firstcol)
PANEL *panel;                /* Panel to be moved */
int firstrow, firstcol;      /* row/col of upper left corner of
                                new location of window associated
                                with panel */
```

Note that the screen coordinates you specify are those for the upper left corner of the window in its new location. The panel may be moved to any location that the low-level ETI routines deem legitimate. In particular, a panel may be partly off the screen. The size, contents, and relative depth of the panel remain unchanged by **move_panel()**.

Function **move_panel()** returns OK if the operation was successful, ERR otherwise. The **move_panel()** operation fails if the low-level ETI functions are unable to move the panel's window, or if there is insufficient memory to satisfy the request. In these cases, the original panel remains unchanged.

To move the panel pointed to by **panel** such that its upper left corner is at row **22**, column **45**, you can write

```
PANEL *panel;

move_panel(panel, 22, 45);
```

Moving Panels to the Top or Bottom of the Deck

The relative depth of a panel can be changed by either pulling the panel to the top of the deck or by pushing it to the bottom. In either case, all other panels remain at the same depth relative to each other.

```
SYNOPSIS

int top_panel(panel)
PANEL *panel;

int bottom_panel(panel)
PANEL *panel;
```

Function **top_panel()** moves the panel pointed to by its argument to the top of the panel deck, while function **bottom_panel()** moves the panel to the bottom of the deck.

Both functions leave the size of the given panel, the contents of its associated window, and the relations of the other panels in the deck wholly intact. Both return OK if the operation is successful, ERR if not. The functions fail if the panel pointer argument is NULL or if the panel is hidden by a previous call to function **hide_panel()** described below.

To move the panel pointed to by **panel1** to the top of the deck of panels and the panel pointed to by **panel2** to the bottom of the deck, you can write the following:

```
PANEL * panel1, * panel2;

top_panel(panel1);
bottom_panel(panel2);
```

Updating Panels on the Screen

Function **update_panels()** makes all low-level **curses** calls (such as **touchwin()** and **wnoutrefresh()**) update all of the panels, so that proper depth relationships are maintained and only appropriate portions of panels are displayed.

```
SYNOPSIS

void update_panels();
```

The function does not, however, actually refresh your terminal screen. To do that, you must make a call to **doupdate()** whenever you want to display your latest changes.

To avoid displaying text on hidden panels, you should not use the low-level routines **wnoutrefresh()** and **wrefresh()** when working with panels.

> | NOTE | In general, do not use the low-level routines **wnoutrefresh()** or **wrefresh()** to display a window associated with a panel. Instead, use function **update_panels()** and function **doupdate()** to display the entire deck of panels.

If you use the low-level routines **wnoutrefresh()** or **wrefresh()** for a window associated with a panel, it will not be displayed properly, unless it happens to be associated with the top panel in the deck or is not hidden at all by other panel windows.

Recall that panels are always above **stdscr**, the standard ETI window. When a panel is moved or deleted, **stdscr** is updated along with the visible panels to ensure that it appears beneath all panels. Although **stdscr** has depth relative to other panels, it is not a panel, because panel operations like **top_panel()** and **bottom_panel()** do not apply. However, because **stdscr** rests beneath the deck of panels, you should always call **update_panels()** when you work with panels and change **stdscr**, even if you do not change any panels.

Function **wgetch()** automatically calls **wrefresh()**. Hence, if echo mode is active, when you request input from a window associated with a panel, be sure that the window is totally unobscured.

In summary, to update all panels and display them with their proper depth relationship, write:

```
WINDOW *win;

update_panels();
doupdate();
```

Finally, note that there is no way to display the updates to an obscured panel without displaying the changes to all panels.

Making Panels Invisible

ETI allows you to hide panels from the deck and later return them to it.

Hiding Panels

Panels may be temporarily hidden. This means that they are removed from the panel deck, but the memory allocated to them is not released.

SYNOPSIS

```
int hide_panel(panel)
PANEL *panel;           /* Pointer to panel to be hidden */
```

Hidden panels are not refreshed to the screen, but you may nonetheless apply nearly all panel operations to them.

> | NOTE | Only the operations **top_panel()**, **bottom_panel()**, and **hide_panel()** may not be applied to hidden panels because their panel arguments must belong to the deck of panels.

When you want to return a hidden panel to the deck of panels, use the function **show_panel()** described in the next section. When the panel is returned, it is placed on top of the deck.

To hide the panel pointed to by **panel2** above, write

```
PANEL *panel2;

hide_panel(panel2);
```

Function **hide_panel()** returns OK if the operation is successful and ERR if its panel pointer argument is NULL.

If you use function **hide_panel()** wisely, your program's performance can increase. You can hide a panel temporarily if no portion of it is to be displayed for awhile. An example is the hiding of a pop-up message. Interim calls to function **update_panels()** will then execute faster. More importantly, you do not incur the overhead of creating the pop-up message.

Checking If Panels are Hidden

To enable you to check if a given panel is hidden, ETI provides the following function.

SYNOPSIS

```
int panel_hidden (panel)
PANEL * panel;
```

Function **panel_hidden()** returns a Boolean value (TRUE or FALSE) indicating whether or not its panel argument is hidden.

You might want to use this function before calling functions **top_panel()** or **bottom_panel()**, which do not operate on hidden panels. For example, to minimize the risk of having the error value ERR returned when moving a panel to the top of the deck, you can write

```
PANEL * panel;

        if (! panel_hidden (panel))    /* panel in deck ? */
                top_panel (panel);  /* move panel to top of deck */
```

Reinstating Panels

This function is the opposite of function **hide_panel()**. It returns the hidden panel referenced in its argument to the top of the panel deck.

SYNOPSIS

```
int show_panel (panel)
PANEL *panel;              /* Panel to return to top of deck */
```

Note that the panel must have been hidden by a previous **hide_panel()** call. The function returns OK if the operation is successful, and ERR if the panel pointer is NULL, if there is insufficient memory, or if the panel is not hidden.

For example, to return **panel2** to the deck, write

```
PANEL * panel2;

show_panel(panel2);
```

Fetching Panels Above or Below Given Panels

The following functions return a pointer to the panel immediately above or below the given panel. They are helpful in walking the panel deck from top to bottom or vice versa.

```
SYNOPSIS


PANEL *panel_above (panel)
PANEL *panel;              /* Get panel above this one */


PANEL *panel_below (panel)
PANEL *panel;              /* Get panel below this one */
```

Because hidden panels have no depth, they are excluded from these traversals.

Function **panel_above()** returns the panel immediately above the given panel. If its argument is NULL, it returns the bottommost panel. The function returns NULL if the given panel is on top or hidden, or if there are no visible panels.

Function **panel_below()** returns the panel immediately below the given panel. If its argument is NULL, it returns the topmost panel. The function returns NULL if the given panel is on the bottom of the deck of panels or hidden, or if there are no visible panels at all. There may be no visible panels at all for the following reasons:

■ They have been hidden using **hide_panel()**

■ All panels have been deleted

■ No panels have been created.

If you want to do something to all panels or to search all of them for one with a particular attribute, you can place one of these functions in a loop. For example, to hide all panels (perhaps to display **stdscr** alone), you can write

```
{
 PANEL *panel, *pnl;

  for (panel = panel_above (NULL); panel; panel = panel_above(pnl))
  {
   pnl = panel;
   hide_panel(panel);
  }
}
```

Setting and Fetching the Panel User Pointer

To enable your application program to associate arbitrary data with a given panel, the ETI panel subsystem automatically allocates a pointer associated with each newly created panel. Initially, the value of this user pointer is NULL. You can set its value to whatever you want or not use it at all.

SYNOPSIS

```
int  *set_panel_userptr (panel, ptr)
PANEL *panel;            /* Panel whose user pointer to set */
char  *ptr;             /* user-defined pointer */

char *panel_userptr (panel)
PANEL *panel;            /* Panel whose user pointer to fetch */
char  *ptr;             /* user-defined pointer */
```

The user pointer has no meaning to the panels subsystem. Once the panel is created, the user pointer is neither changed nor accessed by the subsystem.

Function **set_panel_userptr()** sets the user pointer of a given panel to the value of your choice. The function returns OK if the operation is successful and ERR if the panel pointer is NULL.

Function **panel_userptr()** returns the user pointer for a given panel. If the panel pointer is NULL, the function returns NULL.

You can use these routines to store and retrieve a pointer to an arbitrary structure that holds information for your application. For example, you might use them to store a title or, as in Figure 10-9, create a hidden panel for pop-up messages.

```
PANEL *msg_panel;
char *message = "Pop-up Message Here";  /* initialize message */

int display_deck (show_it)
int show_it;
{
        WINDOW *w;
        int rows, cols;

        if (show_it)
        {       show_panel (msg_panel);  /* reinstate panel */
                w = panel_window (msg_panel);   /* fetch associated window */

                getmaxyx (w, rows, cols); /* fetch window size */

                                            /* center cursor */
                wmove (w, (rows-1), ((cols-1) - strlen(message))/2;

                    /* fetch and write pop-up message */
                    waddstr (w, panel_userptr (msg_panel));
        }
        update_panels();     /* display deck with message, if called for */
        doupdate();
        if (show_it)
                hide_panel (msg_panel); /* hide panel again, if necessary */
}
main()
{
        int show_mess = FALSE;

        msg_panel = new_panel (newwin (10, 10, 5, 60));
        set_panel_userptr (msg_panel, message); /*associate message with panel */
        hide_panel (msg_panel);         /* remove from visible deck */

                                    /* if condition to display pop-up
                                    message is satisfied, set show_mess to TRUE */

        display_deck (show_mess)
}
```

Figure 10-9: Example Using Panel User Pointer

After creating a window and its associated panel, **main()** calls
set_panel_userptr() to set the panel user pointer to point to the panel's pop-
up message string. Function **hide_panel()** hides the panel from the deck so
that it is not normally displayed. Later, the application-defined routine
display_deck() checks if the message is to be displayed. If so, it calls
show_panel() which returns the panel to the deck and enables the panel to
become visible on the next update and refresh. The message string returned
by **panel_userptr()** is then written to the panel window. Finally,
update_panels() adjusts the relative visibility of all panels in the deck, and
doupdate() refreshes the screen. If called for, the pop-up message will now be
visible.

Deleting Panels

The following function deletes a panel, but not its associated window. If you want to delete the window, you should use the low-level function **delwin()**.

```
SYNOPSIS

int del_panel (panel)
PANEL *panel;           /* Panel to be deleted */
```

The ETI panels subsystem assumes that the window associated with each panel always exists.

> NOTE   If you want to delete a panel and its associated window, make sure that you delete the panel first, not the window. Your call to **del_panel()** should precede your call to **delwin()**.

However, it is not necessary to delete a window after its associated panel is deleted; if you like, you may associate the window with another panel.

Function **del_panel()** returns OK if the operation was successful, ERR otherwise. The **del_panel()** operation fails if the panel pointer is NULL.

To delete the panel referenced by **panel** and its associated window referenced by **win**, you can write

```
PANEL * panel;
WINDOW * win = panel_window(panel);

del_panel(panel);
delwin(win);
```
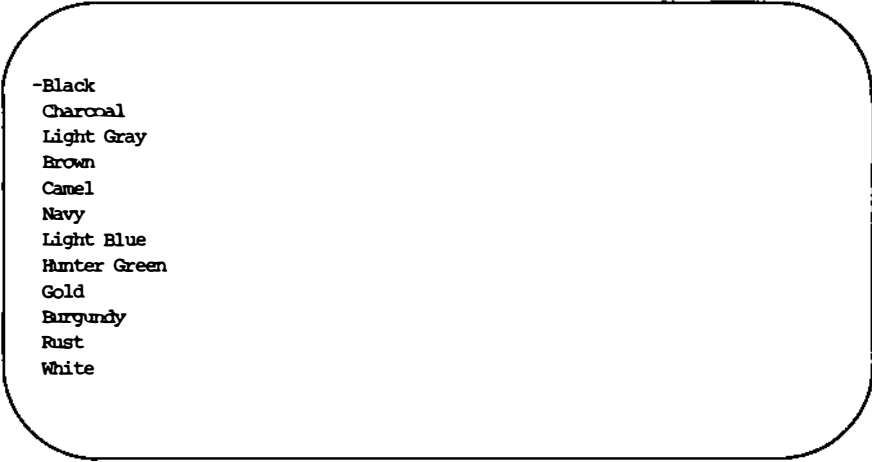
Menus

A menu is a screen display that presents a set of items from which the user selects one or more, depending on the type of menu. Once the user makes a selection, your application program responds accordingly. This response may be to generate a message, display another menu, or take some other action. Figure 10-10 displays a sample menu.

```
 -Black
  Charcoal
  Light Gray
  Brown
  Camel
  Navy
  Light Blue
  Hunter Green
  Gold
  Burgundy
  Rust
  White
```

Figure 10-10: A Sample Menu

Compiling and Linking Menu Programs

To use the menu routines, specify

```
#include <menu.h>
```

in your C program files and compile and link with the command line

cc [*flags*] *files* **-lmenu -lcurses** [*libraries*]

If you use the panel routines as well, specify **-lpanel** before **-lcurses** on the command line.

Overview: Writing Menu Programs in ETI

This section introduces basic ETI menu terminology, lists the steps in a typical menu application program, and reviews the code in a simple example.

Some Important Menu Terminology

The following terms will be helpful:

item	a character string consisting of a name and an optional description
menu	a screen display that presents a set of items from which the user selects one or more, depending on the type of menu
connecting items to a menu	associating an array of item pointers with a menu
menu subwindow	a subwindow on which an associated menu is written
menu window	a window on which an associated menu subwindow and titles and borders, if any, are displayed
posting a menu	writing a menu on its associated subwindow
unposting a menu	erasing a menu from its associated subwindow
pattern matching	checking whether characters entered by the user match an item name of the menu
freeing a menu	deallocating the space for a menu and, as a byproduct, disconnecting an associated array of item pointers from a menu
freeing an item	deallocating the space for an item

NULL   generic term for a null pointer cast to the type of the particular object — item, menu, field, form, and so on

What a Menu Application Program Does

In general, a menu application program will do the following:

- initialize low-level ETI (**curses**)
- create the items for the menu
- create the menu
- post the menu
- refresh the screen
- process end user menu requests
- unpost the menu
- free the menu
- free items
- terminate low-level ETI (**curses**).

A Sample Menu Program

Figure 10-11 shows the ETI code necessary for generating the menu of colors in Figure 10-10.

```
#include <menu.h>

char * colors[13] =
{
        "Black",        "Charcoal",     "Light Gray",
        "Brown",        "Camel",        "Navy",
        "Light Blue",   "Hunter Green", "Gold",
        "Burgundy",     "Rust",         "White",
        (char *) 0
};
```

continued

```
ITEM * items[13];

main ()
{
        MENU *          m;
        ITEM **         i = items;
        char *          c = colors;

        /* low-level ETI (curses) initialization */

        initscr ();
        nonl ();
        raw ();
        noecho ();
        wclear (stdscr);

        /* create items */

        while (*c)
                *i++ = new_item (*c++, "");
        *i = (ITEM *) 0;

        /* create and display menu */

        m = new_menu (i = items);
        post_menu (m);
        refresh;
        sleep (5);

        /* erase menu and free both menu and items */

        unpost_menu (m);
        refresh;
        free_menu (m);

        while (*i)
                free_item (*i++);

        /* low-level ETI (curses) termination */
        endwin ();
        exit (0);
}
```

Figure 10-11: Sample Menu Program to Create a Menu in ETI

To get an overview of ETI menu routines, we will now briefly walk through this menu program. In later sections, we discuss these and remaining ETI routines in detail.

Every menu program should have the line

#include <menu.h>

to instruct the C preprocessor to make the file of ETI menu declarations available. The initial low-level ETI routines establish the best terminal characteristics for working with the ETI menu routines.

The **while** loop creates each item for the menu using the ETI function **new_item()**. This function takes as its name argument a color from array **colors[]**. The optional description argument is here the null string. The new item pointers are assigned to a NULL-terminated array.

Next, the menu is created and the item pointer array is connected to the menu using function **new_menu()**. The menu is then posted to **stdscr** and the screen is refreshed to display the menu. The **sleep()** command makes the menu visible for 5 seconds.

To erase the menu, unpost it and refresh the screen. Function **free_menu** disconnects the menu from its item pointer array and deallocates the space for the menu. The last **while** loop uses function **free_item()** to free the space allocated for each item.

Finally, functions **endwin()** and **exit()** terminate low-level ETI and the program.

The following sections explain how to use all ETI menu routines. Program fragments illustrating the menu routines occur throughout this chapter. Many of these fragments are portions of a larger program example. The current example and others are included in the set of high-level ETI demonstration programs delivered with the ETI product. Low-level ETI demonstration programs are reproduced in the last section of this guide.

> | NOTE | Like all form routines that return an **int** value, all menu routines that do so return the value E_OK when they execute successfully.

Creating and Freeing Menu Items

Normally, to create a menu, you must first create the items comprising it. To create a menu item, use function **new_item()**.

```
SYNOPSIS

ITEM * new_item (name, description)
char * name;
char * description;
```

Function **new_item()** creates a new item by allocating space for the new item and initializing it. ETI displays the string **name** when the menu is later posted, but calling **new_item()** does not alone connect the item to a menu. The item **name** is also used in pattern-matching operations. If **name** is NULL or the null string, then **new_item()** returns NULL to indicate an error.

The argument **description** is a descriptive string associated with the item. It may or may not be displayed depending on the O_SHOWDESC option, which you can turn on or off with the **set_menu_opts()** and related functions described below. If **description** is NULL or the null string, no description is associated with the menu item.

If successful, **new_item()** returns a pointer to the new item. This pointer is the key to working with all item routines. When you pass it to them, it enables the menu subsystem to change, record, and examine the item's attributes.

If there is insufficient memory for the item, or **name** is NULL or the null string, then **new_item()** returns NULL.

In general, use an array to store the item pointers returned by **new_item()**. Figure 10-12 shows how you might create an item array of the planets of our solar system.

```
ITEM * planets[10];

planets[0] = new_item ("Mercury", "The first planet");
planets[1] = new_item ("Venus", "The second planet");
planets[2] = new_item ("Earth", "The third planet");
planets[3] = new_item ("Mars", "The forth planet");
planets[4] = new_item ("Jupiter", "The fifth planet");
planets[5] = new_item ("Saturn", "The sixth planet");
planets[6] = new_item ("Uranus", "The seventh planet");
planets[7] = new_item ("Neptune", "The eighth planet");
planets[8] = new_item ("Pluto", "The ninth planet");
planets[9] = (ITEM *) 0;
```

Figure 10-12: Creating an Array of Items

Function **new_item()** does not copy the name or description strings, but saves the pointers to them. So once you call **new_item()**, you should not change the strings until you call **free_item()**.

SYNOPSIS

```
free_item(item);
ITEM * item;
```

Function **free_item()** frees an item. It does not, however, deallocate the space for the item's name or description.

The argument to **free_item()** is a pointer previously obtained from **new_item()**.

NOTE To free an item, you must have already created it with **new_item()** and it must not be connected to a menu. If these conditions are not met, **free_item()** returns one of the error values listed below.

Once an item is freed, you must not use it again. If a freed item's pointer is

passed to an ETI routine, undefined results will occur.

If successful, **free‑item()** returns E‑OK. If it encounters an error, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null item
E_CONNECTED	- item is connected to a menu

Two Kinds of Menus: Single- and Multi-Valued

Menus are of two kinds:

single-valued menus   from which the user may select only one item

multi-valued menus   from which the user may select one or more items

By default, every menu is single-valued. To create a multi-valued menu, you turn off menu option O_ONEVALUE using function **set_menu_opts()** or **menu_opts_off()**. These functions are treated in the section , "Setting Item Options."

Menus of both types always have a current item. With single-valued menus, you determine the item selected by noting the current item. With multi-valued menus, you determine all items selected by applying function **item_value()** to each menu item and noting the value returned. Most menu functions pertain to menus whether they are single- or multi-valued. Function **set_item_value()**, however, may be used only with multi-valued menus.

Manipulating an Item's Select Value in a Multi-Valued Menu

Select values of an item are either TRUE (selected) or FALSE (not selected). Function **set_item_value()** sets the select value of an item, while **item_value()** returns it.

```
SYNOPSIS

int set_item_value (item, value)
ITEM * item;
int value;

int item_value (item)
ITEM * item;
```

Function **set_item_value()** fails if given an item that is not selectable (the O_SELECTABLE option was previously turned off) or the item is connected to a single-valued menu (connecting items to menus is described in the section,

"Creating and Freeing Menus"). If successful, **set_item_value()** returns E_OK. Otherwise, one of the following is returned.

> E_SYSTEM_ERROR   - system error
> E_REQUEST_DENIED   - item not selectable or single value menu

If the argument to **item_value()** is an item pointer connected to a single-valued menu, **item_value()** returns FALSE.

You might want to place the code in Figure 10-13 after your user responds to a menu. Function **process_menu()** determines which items have been selected, processes them appropriately, and marks them as unselected to prepare for further user response.

```
void process_menu (m) /* process multi-valued menu */
MENU * m;
{
        ITEM ** i = menu_items (m);

        while (*i) {
        {
                if (item_value (*i)) {
                {

                        /* take action appropriate for selection of this item */

                        set_item_value (*i, FALSE);
                }
                ++i;
        }
}
```

Figure 10-13: Using **item_value()** in Menu Processing

Manipulating Item Attributes

An attribute is any feature whose value can be set or read by an appropriate ETI function. An item attribute is any item feature whose value can be set or read by an appropriate ETI function. Item names, descriptions, options, and visibility are examples of item attributes.

Fetching Item Names and Descriptions

The routines **item_name()** and **item_description()** take an item pointer as their argument. Function **item_name()** returns the item's name, while function **item_description()** returns its description.

```
SYNOPSIS

char * item_name (item)
ITEM * item;

char * item_description (item)
ITEM * item;
```

Both functions return NULL if given a NULL item pointer.

Setting Item Options

An option is an attribute whose value may be either on or off. The current release of ETI provides the item option O_SELECTABLE. (In the future, ETI may provide additional options.) Setting the O_SELECTABLE option lets your user select the item. By default, O_SELECTABLE is set for every item. Function **set_item_opts()** lets you turn on or turn off this and any future options for an item, while **item_opts()** lets you examine the option(s) set for a given item.

```
SYNOPSIS

int set_item_opts (item, opts)
ITEM * item;
OPTIONS opts;

OPTIONS item_opts (item)
ITEM * item;

options:
        O_SELECTABLE
```

In addition to turning on the named item options, function **set_item_opts()** turns off any other item options.

If successful, **set_item_opts()** returns E_OK. Otherwise, it returns the following:

      E_SYSTEM_ERROR   - system error

If function **set_item_opts()** is passed a NULL item pointer, like other functions it sets the new current default. If function **item_opts()** is passed a NULL pointer, it returns the current default.

If you turn off option O_SELECTABLE, the item cannot be selected. You might want to make an item unselectable to emphasize certain things your application program is doing. Unselectable items are displayed using the grey display attribute, described below in the section "Setting Menu Display Attributes."

Because options are Boolean values (they are either on or off), you must use C Boolean operators with **item_opts()** to turn them on and off. Consequently, to turn off option O_SELECTABLE for item **i0** and turn on the same option for item **i1**, write the following:

```
ITEM * i0, * i1;

set_item_opts (i0, item_opts (i0) & ~O_SELECTABLE); /* turn option off */
set_item_opts (i1, item_opts (i1) |  O_SELECTABLE); /* turn option on  */
```

ETI also enables you to turn on and off specific item options without affecting others, if any. The following functions change only the options specified.

```
SYNOPSIS

int item_opts_on (item, opts)
ITEM * item;
OPTIONS opts;

int item_opts_off (item, opts)
ITEM * item;
OPTIONS opts;
```

These functions return the same error conditions as **set_item_opts()**.

For example, the following code turns option O_SELECTABLE off for item **i0** and on for item **i1**.

```
ITEM * i0, * i1;

item_opts_off (i0, O_SELECTABLE); /* turn option off */

item_opts_on  (i1, O_SELECTABLE); /* turn option on  */
```

To change the current default to not O_SELECTABLE, you can write either

```
            /* set current defaults for all new items */

set_item_opts ((ITEM *) 0, item_opts( (ITEM *) 0) & ~O_SELECTABLE);
```

or

```
item_opts_off ((ITEM *) 0, O_SELECTABLE); /* turn default option off */
```

Checking an Item's Visibility

A menu item is visible if it appears in the subwindow of the posted menu to which it is connected. (Connecting and Posting Menus is described below.) Function **item_visible()** enables your application program to determine if an item is visible.

SYNOPSIS

```
int item_visible (item)
ITEM * item;
```

If the item is connected to a posted menu and it appears in the menu subwindow, **item_visible()** returns TRUE. Otherwise, it returns FALSE.

To check if the first menu item is currently visible on the display, write

```
int at_top (m) /* check visibility of first menu item */
MENU * m;
{
        ITEM ** i = menu_items (m);
        ITEM * firstitem = i[0];

        return item_visible (firstitem);
}
```

For another example, see the section, "Counting the Number of Menu Items."

Changing the Current Default Values for Item Attributes

ETI establishes initial current default values for item attributes. During item initialization, each item attribute is assigned the current default value of the attribute. You can change or retrieve the current default attribute values by calling the appropriate function with a NULL item pointer. After the current default value changes, all subsequent items created with **new_item()** will have the new default value.

> NOTE — Items created before changing the current default value retain their previously assigned values.

The following sections offer many examples of how to change item attributes.

Setting the Item User Pointer

For each item created, ETI automatically allocates a special user pointer that enables you to associate arbitrary data with the item. By default, the user pointer's value is NULL. You may set its value to whatever you want or not use it at all.

```
SYNOPSIS

int set_item_userptr (item, userptr)
ITEM * item;
char * userptr;

char * item_userptr (item)
ITEM * item;
```

These two functions are helpful for creating item data such as title strings, help messages, and the like.

Any defined structure can be connected to an item using the item's user pointer. The pointer must be cast to (char *) and then later recast back to (defined-struct *). Figure 10-14 shows how to use an item's user pointer with a struct ITEM_ID, which stores biological information.

```
typedef struct
{
        int     id;
        char *  name;
        char *  type;
}
        ITEM_ID;

ITEM_ID ids[7] =
{
        1, "apple", "fruit",
        2, "ant", "insect",
        3, "cow", "mammal",
        4, "lizard", "reptile",
        5, "potato", "vegetable",
        6, "zebra", "mammal",
        0, "", "",
};

ITEM * items[7];

for (i = 0; ids[i]; ++i)
{
        /* create item from each ids.name */

        items[i] = new_item (ids[i].name, "");

        /* set user pointer to point to start of each struct in ids[] */

        set_item_userptr (items[i], (char *) &ids[i]);
}
items[i] = (ITEM *) 0;
```

Figure 10-14: Using an Item User Pointer

Note that the pointer to each entry in array **ids** is cast to **char ***, which
set_userptr() requires. You might then write a function that uses function
item_userptr() to return the information. The following function returns the

item type.

```
char * get_type (i)
ITEM * i;
{
        ITEM_ID * id = (ITEM_ID *) item_userptr (i);
        return id -> type;
}
```

Here, the value returned by **item_userptr()** is recast to **ITEM_ID *** so the item's **type** may be found.

Finally, you might call **get_type()** to write the type, thus:

```
WINDOW * win;

waddstr (win,get_type(i));
```

If successful, **set_item_userptr()** returns E_OK. Otherwise, it returns the following:

```
E_SYSTEM_ERROR          - system error
```

If function **set_item_userptr()** is passed a NULL item pointer, the argument **userptr** becomes the new default user pointer for all subsequently created items. For example, the following sets the new default user pointer to point to the string "You are Here":

```
set_item_userptr( (ITEM *) 0, "You are Here");
```

Creating and Freeing Menus

Once you create the items for your menu, you can create the menu. To create and initialize a menu, use function **new_menu()**.

```
SYNOPSIS

MENU * new_menu (items)
ITEM ** items;
```

The argument to **new_menu()** is a NULL terminated, ordered array of ITEM pointers. These pointers define the items on the menu. Their order determines the order in which the items are visited during menu driver processing, described below.

Function **new_menu()** does not copy the array of item pointers. Instead, it saves the pointer to the array for future use.

| NOTE | Once your application program has called **new_menu()**, it should not change the array of item pointers until the menu is freed by **free_menu()** or the item array is replaced by **set_menu_items()**, as described below. |

Items passed to **new_menu()** are connected to the menu created. They cannot be simultaneously connected to another menu. To disconnect the items from a menu, you can use function **free_menu()** or function **set_menu_items()**, which changes the items connected to a menu from one set to another. See the section "Fetching and Changing Menu Items".

If successful, **new_menu()** returns a pointer to the new menu. The following error conditions hold:

■ If there is insufficient memory for the menu or it detects an item connected to another menu, **new_menu()** returns NULL.

■ If the array of item pointers is not NULL-terminated, undefined results occur.

In addition, if **new_menu()**'s argument **items** is NULL, as in

```
MENU * m;

m = new_menu ((MENU *) 0);
```

it creates the menu with no items connected to it and assigns the menu
pointer to **m**.

The menu pointer returned by **new_menu()** is the key to working with all
menu routines. Pass it to the appropriate menu routine to do such tasks as
post menus, call the menu driver, set the current item, and record or examine
menu attributes.

Turn again to Figure 10-11 for an example of how to create a menu. In
general, you want to use a **while** loop as illustrated to create the menu items
and assign the item pointers to the item pointer array. Note the NULL termi-
nator assigned to the item pointer array before the menu is created with
new_menu().

When you no longer need a menu, you should free the space allocated for
it. To do this, use function **free_menu()**.

```
SYNOPSIS

int free_menu (menu)
MENU * menu;
```

Function **free_menu()** takes as its argument a menu pointer previously
obtained from **new_menu()**. It disconnects all items from the menu and frees
the space allocated for the menu. The items associated with the menu are not
freed, however, because you may want to connect them to another menu. If
not, you can free them by calling **free_item()**.

Remember that once a menu is freed, you must not pass its menu pointer
to another routine. If you do, undefined results occur.

If successful, calls to **free_menu()** return E_OK. If **free_menu()** encounters an error, it returns one of the following:

E_BAD_ARGUMENT	- NULL menu pointer
E_POSTED	- menu is posted
E_SYSTEM_ERROR	- system error

For E_POSTED, see the section, "Posting and Unposting Menus."

Manipulating Menu Attributes

Recall that an attribute is any feature whose value can be set or read by an appropriate ETI function. A menu attribute is any menu feature whose value can be set or read by an appropriate ETI function. The set of items connected to a menu and the number of items in the menu are examples of menu attributes.

Fetching and Changing Menu Items

During processing, you may sometimes want to change the set of items connected to a menu. Function **set_menu_items()** enables you to do this.

```
SYNOPSIS

int set_menu_items (menu, items)
MENU * menu;
ITEM ** items;

ITEM ** menu_items (menu)
MENU * menu;
```

Like the argument to **new_menu()**, the second argument to **set_menu_items()** is a NULL-terminated, ordered array of ITEM pointers that defines the items on the menu. Like **new_menu()**, function **set_menu_items()** does not copy the array of item pointers. Instead, it saves the pointer to the array for future use.

The items previously connected to the given menu when **set_menu_items()** is called are disconnected from the menu (but not freed) before the new items are connected. The new items cannot be given to other menus unless first disconnected by **free_menu()** or another **set_menu_items()** call.

If **items** is NULL, the items associated with the given menu are disconnected from it, but no new items are connected.

If function **set_menu_items()** is successful, it returns E_OK. If it encounters an error, it returns one of the following:

E_SYSTEM_ERROR - system error
E_BAD_ARGUMENT - NULL menu pointer or NULL associated item array
E_POSTED - menu is posted
E_CONNECTED - connected item

Function **menu_items()** returns the array of item pointers associated with its menu argument. In the next section, the application-defined function **at_bottom()** illustrates its use.

If no items are connected to the menu or the menu pointer argument is NULL, **menu_items()** returns NULL.

As an example of **set_menu_items()**, consider Figure 10-15, whose code changes the items associated with a previously created menu.

```
MENU *m;
ITEMS ** olditems, ** newitems;
                              /* create items */


m = new_menu(olditems);      /* create menu m */

                             /* process menu with olditems */

set_menu_items (m,newitems); /* change items associated with menu m */
```

Figure 10-15: Changing the Items Associated With a Menu

Counting the Number of Menu Items

Occasionally, you may want to do different processing, depending on the number of items connected to your current menu. Function **item_count()** returns the number of items connected to a menu.

```
SYNOPSIS

int item_count (menu)
MENU * menu;
```

If **menu** is NULL, function **item_count()** returns -1.

As an example of the use of this function, consider the following routine. Because the index to the last menu item is one less than the number of items, this routine determines whether the last item is displayed.

```
int at_bottom (m) /* check visibility of last menu item */
MENU * m;
{
        ITEM ** i = menu_items (m);
        ITEM * lastitem = i[item_count(m)-1];

        return item_visible (lastitem);
}
```

Changing the Current Default Values for Menu Attributes

As it does with the attributes of other objects, ETI establishes initial current default values for menu attributes. During menu creation, each menu attribute is assigned the current default value of the attribute. You can change or retrieve the current default attribute values by calling the appropriate function with a NULL menu pointer. After the current default value changes, all subsequent menus created with **new_menu()** will have the new default value.

| NOTE | Menus created before changing the current default value retain their pre-viously assigned values. |

The following sections offer many examples of how to change menu attributes.

Displaying Menus

In general, to display a menu, you must determine the menu's dimensions, optionally associate a window and subwindow with the menu, optionally set the menu's display attributes, post the menu, and refresh the screen.

Determining the Dimensions of Menus

The simplest way to display a menu is to use **stdscr** as your default window and subwindow. Any titles, borders, or other decorative matter are displayed in the menu window; the menu proper is displayed in the menu subwindow. If you want to specify a menu window or subwindow, use the functions **set_menu_win()** or **set_menu_sub()**. (These routines are treated in the section, "Associating Windows and Subwindows with Menus".) Whether or not you choose a menu window, ETI calculates the minimum window (or subwindow) size for your menu.

To determine the minimum window size for a menu, ETI considers five factors:

- the size and number of items in a menu

- whether option O_ROWMAJOR is on

- whether option O_SHOWDESC is on

- the format, or maximum number of rows and columns on a displayed page of the menu

- the mark string for menu items

ETI knows the size and number of items in a menu as soon as you call **new_menu()**, discussed above. By default, options O_ROWMAJOR and O_SHOWDESC are on. Option O_ROW_MAJOR ensures that the items are displayed in row major order — fanning out left to right, then top to bottom. Option O_SHOWDESC ensures that an item's description, if any, is displayed with the item's name.

This section first describes the menu's format and mark string. It then describes the routine **scale_menu()**, which uses the above information to set the window size for the menu.

| NOTE | The five factors that determine the minimum window size have default values. You need not worry about them until you want to customize your menus. |

Specifying the Menu Format

In general, the items comprising a menu do not fill a single screen. Sometimes they occupy considerably less space, sometimes considerably more. The following functions enable you to set the maximum number of rows and columns of menu items to be displayed at any one time.

```
SYNOPSIS

int set_menu_format (menu, maxrows, maxcols)
MENU * menu;
int maxrows, maxcols;

void menu_format (menu, maxrows, maxcols)
MENU * menu;
int * maxrows, * maxcols;
```

A menu page is the collection of currently visible items. Function **set_menu_format()** establishes the maximum number of rows and columns of items that may be displayed on a menu page.

The actual number of rows and columns displayed may be less than **maxrows** or **maxcols** depending on the number of items and whether the O_ROWMAJOR option is on. (Menu options are described in the section, "Setting and Fetching Menu Options".) Function **menu_format()** returns the maximum number of rows and columns of items that you set for the given menu.

The default number of item rows is 16, while the default number of item columns is 1. If either **maxrows** or **maxcols** equals 0 in the call to **set_menu_format()**, the current value is not changed. An error occurs, however, if the value of either of these arguments is less than 0.

   ETI calculates the total number of rows and columns in a row major menu as follows:

```
#define minimum(a,b)((a) < (b) ? (a) : (b))


total_rows = (number_of_items - 1) / maxcols + 1;
total_cols = minimum (number_of_items, maxcols);
```

   ETI calculates the total number of rows and columns in a column major menu as follows:

```
total_rows = (number_of_items - 1) /maxcols + 1;
total_cols = (number_of_items - 1) / total_rows + 1;
```

   Whether or not the O_ROW_MAJOR option is on, the number of rows and columns of items that are displayed at one time on a menu page is

```
displayed_rows = minimum (total_rows, maxrows);
displayed_cols = minimum (total_cols, maxcols);
```

   If **total_rows** is greater than **maxrows**, the menu is scrollable — your end-user can scroll up or down through the menu by making the appropriate menu driver request. See the section, "Menu ETI Requests".

   As an example, consider the displays in Figures 10-16 and 10-17. They portray menus consisting of 5 items. The numbers 0 through 4 signify menu items in the order in which they live in the item pointer array. Figure 10-16 shows the menu displayed with a format of maximum number of rows 2, maximum number of columns 2. To stipulate this format for menu **m**, write

```
set_menu_format(m,2,2);
```

Using the formulas above, we see that **total_rows** is 3 and **total_cols** is 2 in all four cases displayed in the two figures. The first display in each figure shows the menu in row-major format (O_ROW_MAJOR on), the second in column-major format. The displayed number of rows and columns in Figure 10-16 is 2. To see the last row of items, your user can make the REQ_SCR_DLINE request to scroll down. If, instead, you set the format of this menu to 3 rows, 2 columns, you get 1 of the 2 displays in Figure 10-17. The enclosing block in each case indicates the items displayed at one time.

```
┌───────────┐              ┌───────────┐
│  ─        │              │ 0      3  │
│ 0    1    │              │ 1      4  │
│ 2    3    │              └───────────┘
└───────────┘
 4                          2
```

Row Major   Column Major

Maximum Rows 2
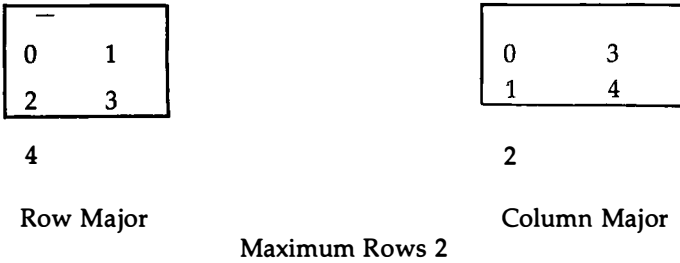
Figure 10-16: Examples of Menu Format (2, 2)

```
┌───────────┐              ┌───────────┐
│  ─        │              │ 0      3  │
│ 0    1    │              │ 1      4  │
│ 2    3    │              │ 2         │
│ 4         │              └───────────┘
└───────────┘
```

Row Major   Column Major

Maximum Rows 3
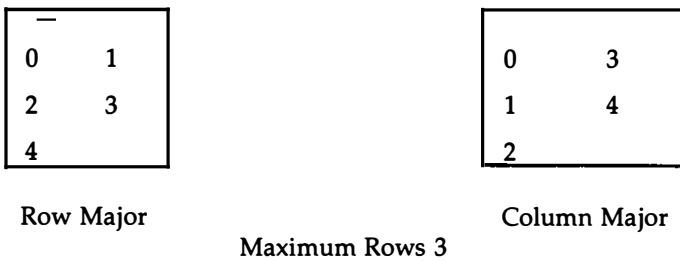
Figure 10-17: Examples of Menu Format (3, 2)

For a larger example, consider Figure 10-18. Here the number of items is 18 and the format in both cases is four rows, three columns. In both cases, the total number of rows comes to 6, the total number of columns to 3, and the displayed number of rows to 4. Calculation shows that changing the number of items in this example to 19 changes the number of rows to 7.

0	1	2
3	4	5
6	7	8
9	10	11

12	13	14
15	16	17

0	6	12
1	7	13
2	8	14
3	9	15

4	10	16
5	11	17

Row Major   Column Major

Figure 10-18: Examples of Menu Format (4, 3)

───

The column major examples emphasize that when the total number of rows is greater than the maximum number of rows, the items displayed do not exactly follow the order of the items in the array of item pointers. The items are arranged in column-major format throughout the entire menu, not within each displayed page. This conception agrees with your user's ability to scroll through the menu.

If successful, function **set_menu_format()** returns E_OK. If an error occurs, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- rows < 0 or cols < 0
E_POSTED	- menu is posted

If function **set_menu_format()** is passed a NULL menu pointer, it sets a new system default. Suppose, for instance, that you want to change the default maximum number of rows of items displayed to 10, and the default maximum number of columns displayed to 3. You can write

```
set_menu_format((MENU *)0,10,3);
```

The function **set_menu_format()** resets the value of **top_row()** to 0. See the section, "Fetching and Changing the Top Row," for details.

Finally, if function **menu_format()** receives a NULL menu pointer, it returns the current default format.

Changing Your Menu's Mark String

The mark string distinguishes

■ selected items in a multi-valued menu

■ the current item in a single-valued menu

The mark string appears just to the left of the item name.

```
SYNOPSIS

int set_menu_mark (menu, mark)
MENU * menu;
char * mark;


char * menu_mark (menu)
MENU * menu;
```

Function **set_menu_mark()** sets the mark string, while **menu_mark()** returns the string. The initial default mark string is a minus sign ("-"). The mark string may be as long as you want, provided each item fits on one line of the menu's subwindow.

> NOTE | Do not change the mark string area as long as you want that mark, because ETI does not copy it.

If **mark** is NULL, no mark string appears.

You can call **set_menu_mark()** either before or after the menu is posted. (See the section, "Posting and Unposting Menus.") However, there is a restriction to calling it afterwards.

> NOTE | If you call **set_menu_mark()** with a posted menu, the length of the mark string must stay the same.

If the menu is posted and the length of the mark string changes, the function returns E_BAD_ARGUMENT and leaves the mark unchanged.

To change the mark string for menu **m** to `"--->"`, you can write

```
MENU * m;

set_menu_mark (m, "--->"); /* change mark string for menu m */
```

If successful, function **set_menu_mark()** returns E_OK. If an error occurs, function **set_menu_mark()** returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- menu is posted: change in string length impossible

Note that you can change the current default mark string for all subsequently created menus in your program by passing **set_menu_mark()** a NULL menu pointer. To change the current default mark string to `"--->"`, write

```
set_menu_mark ((MENU *) 0, "--->"); /* change default mark string */
```

All subsequently created menus will have `"--->"` as their mark string. To return the current default mark string, call **menu_mark()** with NULL:

```
char * mark = menu_mark ((MENU *) 0); /* default mark string */
```

Querying the Menu Dimensions

Remember that the size of menu items, the O_ROWMAJOR menu option, the menu format, and the menu mark determine the smallest window size for a menu. Function **scale_menu()** returns this smallest window size in terms of the number of character rows and columns.

```
SYNOPSIS

int scale_menu (menu, rows, cols)
MENU * menu;
int * rows, * cols;
```

Because function **scale_menu()** must return more than one value (namely, the minimum number of rows and columns for the menu) and C passes

parameters "by value" only, the arguments of **scale_menu()** are pointers. The pointer arguments **rows** and **cols** point to locations used to return the minimum number of rows and columns for displaying the menu.

| NOTE | You should call **scale_menu()** only after the menu's items have been connected to the menu using **new_menu()** or **set_menu_items()**. |

The following code places the minimal number of rows and columns necessary for menu **m** in **rows** and **cols**:

```
MENU *m;
int rows, cols;

scale_menu (m, &rows, &cols); /* return dimensions of menu m */
```

You use the values returned from **scale_menu()** to create menu windows and subwindows. In the next section, we will see how to do this.

If successful, **scale_menu()** returns E_OK. If an error occurs, the function returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null menu pointer
E_NOT_CONNECTED	- no connected items

Associating Windows and Subwindows with Menus

Two windows are associated with each menu—the menu window and the menu subwindow. The following functions assign windows and subwindows to menus and fetch those previously assigned to them.

SYNOPSIS

int set_menu_win (menu, window)
MENU * menu;
WINDOW * window;

WINDOW * menu_win (menu)
MENU * menu;

int set_menu_sub (menu, window)
MENU * menu;
WINDOW * window;

WINDOW * menu_sub (menu)
MENU * menu;

To place a border around your menu or give it a title, call **set_menu_win()** and write to the associated window.

| NOTE | By default, (1) the menu window is NULL, which by convention means that ETI uses **stdscr** as the menu window; and (2) the menu subwindow is NULL, which means that ETI uses the menu window as the menu subwindow. |

If you do not want to use the system defaults, you may create a window and a subwindow for every menu. ETI automatically writes all output of the menu proper on the menu's subwindow. You may write additional output (such as borders, titles, and the like) on the menu's window. The relationship between ETI menu routines, your application program, a menu window, and a menu subwindow is illustrated in Figure 10-19.
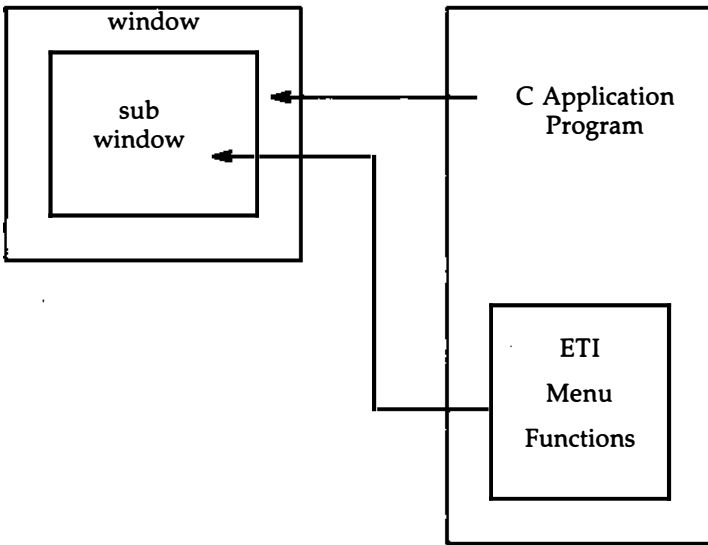
Figure 10-19: Menu Functions Write to Subwindow, Application to Window

---

NOTE   You should apply all output and refresh operations to the menu window, not its subwindow.

Figure 10-20 shows how you can create and display a menu with a border of the default characters, ACS_VLINE and ACS_HLINE. (See the entry on the **box** command in the **curses**(3X) manual page.)

```
MENU * m;
WINDOW * w;
int rows, cols;

scale_menu (m, &rows, &cols); /* get dimensions of menu */

        /* create window 2 characters larger than menu dimensions
        with top left corner at (0, 0). subwindow is positioned
        at (1, 1) relative to menu window origin with dimensions
        equal to the menu dimensions.   */

if (w = newwin (rows+2, cols+2, 0, 0))
{
        set_menu_win (m, w);
        set_menu_sub (m, derwin (w, rows, cols, 1, 1));

        box (w, 0, 0); /* draw border in w */
}
```

Figure 10-20: Creating a Menu with a Border

Variables **rows** and **cols** provide the menu dimensions without the border. The dimensions of the menu subwindow are set to these values. In general, if you want a simple border, you should set the number of rows and columns in the menu's window to be two more than the numbers in its subwindow, as in the example.

Remember that the initial default menu window and subwindow are NULL. (By convention, this means that **stdscr** is used as the menu window and the menu window is used as the menu subwindow.) If you want to change the current default menu window or subwindow, you can pass functions **set_menu_win()** and **set_menu_sub()** a NULL menu pointer. Thus, the code

```
WINDOW * dftwin;

set_menu_win ((MENU *) 0, dftwin); /* sets default menu window
to dftwin */
```

changes the current default window to **dftwin**.

If successful, functions **set_menu_win()** and **set_menu_sub()** return E_OK. If not, they return one of the following:

E_SYSTEM_ERROR   - system error
E_POSTED   - menu is posted

Fetching and Changing A Menu's Display Attributes

Menu display attributes are visible menu characteristics that distinguish classes of menu items from each other. Low-level ETI (**curses**) color and video attributes are used to differentiate the menu display attributes. These menu display attributes include

foreground attribute   distinguishes the current item, if selectable, on all menus and selected items on multi-valued menus

background attribute   distinguishes selectable, but unselected, items on all menus

grey attribute   distinguishes unselectable items on multi-valued menus

pad character   the character that fills (pads) the space between a menu item's name and description

The following functions enable you to set and read these attributes.

SYNOPSIS

```
int set_menu_fore (menu, attr)
MENU * menu;
chtype attr;

chtype menu_fore (menu)
MENU * menu;

int set_menu_back (menu, attr)
MENU * menu;
chtype attr;

chtype menu_back (menu)
MENU * menu;

int set_menu_grey (menu, attr)
MENU * menu;
chtype attr;

chtype menu_grey (menu)
MENU * menu;

int set_menu_pad (menu, pad)
MENU * menu;
int pad;

int menu_pad (menu)
MENU * menu;
```

In general, to establish uniformity throughout your program, you should set the menu display attributes with these functions at the start of the program.

Function **set_menu_fore()** sets the **curses** foreground attribute. The default is A_STANDOUT.

Function **set_menu_back()** sets the **curses** background attribute. The default is A_NORMAL.

Function **set_menu_grey()** sets the **curses** attribute used to display non-selectable items. The default is A_UNDERLINE.

To set the foreground attribute of menu **m** to A_BOLD and its background attribute to A_DIM, you write

```
MENU *m;

set_menu_fore(m,A_BOLD);
set_menu_back(m,A_DIM);
```

All these functions can change or fetch the current default if passed a NULL menu pointer. For example, to set the default grey attribute to A_NORMAL, write

```
set_menu_grey((MENU *)0, A_NORMAL);
```

If functions **set_menu_fore()**, **set_menu_back()**, and **set_menu_grey()** encounter an error, they return one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- bad **curses** attribute

Function **set_menu_pad()** sets the pad character for a menu. The initial default pad character is a blank. The pad character must be a printable ASCII character.

To change the pad character for menu **m** to a dot ('.'), write

```
MENU * m;

set_menu_pad(m,'.');
```

If function **set_menu_pad()** encounters an error, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- nonprintable pad character

Posting and Unposting Menus

To post a menu is to write it on the menu's subwindow. To unpost a menu is to erase it from the menu's subwindow, but not destroy its internal data structure. ETI provides two routines for these actions.

```
SYNOPSIS

int post_menu (menu)
MENU * menu;

int unpost_menu (menu)
MENU * menu;
```

Note that neither of these functions actually change what is displayed on the terminal. After posting or unposting a menu, you must call **wrefresh()** (or its equivalents, **wnoutrefresh()** and **doupdate()**) to do so.

If function **post_menu()** encounters an error, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null menu pointer
E_POSTED	- menu is already posted
E_NOT_CONNECTED	- no connected items
E_NO_ROOM	- menu does not fit in subwindow

Regarding E_NO_ROOM, recall from the section, "Querying the Menu Dimensions," that function **scale_menu()** returns the number of rows and columns necessary to display the menu. It does not, however, know the size of the subwindow you are associating with the menu. Only when the menu is posted is this point checked. Any failure of the menu to fit in the subwindow is then detected.

If function **unpost_menu()** executes successfully, it returns E_OK. In the following situations, it fails and returns the indicated values:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null menu pointer
E_NOT_POSTED	- menu is not posted
E_BAD_STATE	- called from init or term

You might, for instance, receive E_NOT_POSTED if you forgot to post the menu in the first place or you mistakenly tried to unpost it twice.

Figure 10-21 illustrates two routines you might write to post and unpost menus. Function **display_menu()** creates the window and subwindow for the menu and posts it. Most of its code we saw previously in Figure 10-20. Function **erase_menu()** unposts the menu and erases its associated window and subwindow.

```
static void display_menu (m)      /* create menu windows and post */
MENU * m;
{
        WINDOW * w;
        int          rows;
        int          cols;

        scale_menu (m, &rows, &cols);  /* get dimensions of menu */

                /* create menu window, subwindow, and border */

        if (w = newwin (rows+2, cols+2, 0, 0)) {
                set_menu_win (m, w);
                set_menu_sub (m, derwin (w, rows, cols, 1, 1));
                box (w, 0, 0);     /* create border of 0's */
                keypad (w, 1);     /* set for each data entry window */
        }
        else
                error ("error return from newwin", NULL);

                /* post menu */

        if (post_menu (m) != E_OK)
                error ("error return from post_menu", NULL);
        else
                wrefresh (w);
}
static void erase_menu (m)        /* unpost and delete menu windows */
MENU * m;
{
        WINDOW * w = menu_win (m);
        WINDOW * s = menu_sub (m);

        unpost_menu (m);       /* unpost menu */
        werase (w);            /* erase menu window */
        wrefresh (w);          /* refresh screen */
        delwin (s);     /* delete menu windows */
        delwin (w);
}
```

Figure 10-21: Sample Routines Displaying and Erasing Menus

Function **keypad()** is called with a second argument of **1** to enable virtual keys KEY_LL, KEY_LEFT, and others to be properly interpreted in the routine **get_request()** described in, "Menu Driver Processing." See the discussion of **keypad()** in the **curses**(3X) manual page for details. Finally, note the placement of checks for error returns in this example.

Menu Driver Processing

The **menu_driver()** is the workhorse of the menu system. Once the menu is posted, the **menu_driver()** handles all interaction with the end user. It responds to the following:

- item navigation requests

- menu scrolling requests

- item selection requests

- pattern buffer requests.

```
SYNOPSIS

int menu_driver (menu, c)
MENU * menu;
int c;
```

Your application program passes a character to the menu driver for processing and evaluates the results.

To enable your application program to fetch the character for the menu driver, write a routine that defines the input key virtualization. This is the correspondence between specific input keys, control characters, or escape sequences on the one hand and menu driver requests on the other. The virtualization routine returns a specific menu request or application command that the menu driver can process. Upon return from the menu driver, your application can check if the input was processed appropriately. If not, your application specifies the action to be taken. These actions may include terminating interaction with the menu, responding to help requests, generating an error message, and so forth.

Defining the Key Virtualization Correspondence

To illustrate a key virtualization routine, consider Figure 10-22, which shows the key virtualization routine **get_request()**. Nearly all the values it returns are the ETI menu requests to be discussed in the following sections.

```
          /* define application commands */

#define QUIT          (MAX_COMMAND + 1)

          /* Note that ^X represents the character control-X.

                ^Q           - end menu processing

                ^N           - move to next item
                ^P           - move to previous item
                home key     - move to first item
                home down    - move to last item

                left arrow   - move left to item
                right arrow  - move right to item
                down arrow   - move down to item
                up arrow     - move up to item

                ^U           - scroll up a line
                ^D           - scroll down a line
                ^B           - scroll up a page
                ^F           - scroll down a page

                ^X           - clear pattern buffer
                ^H <BS>      - delete character from pattern buffer
                ^A           - request next pattern match
                ^Z           - request previous pattern match

                ^T           - toggle item    */

static int get_request (w)    /* virtual key mapping */
WINDOW * w;
{
        int c = wgetch (w);   /* read a character */

        switch (c)
        {
                case 0x11:    /* ^Q */return  QUIT;

                case 0x0e:    /* ^N */return  REQ_NEXT_ITEM;
                case 0x10:    /* ^P */return  REQ_PREV_ITEM;
                case KEY_HOME:      return  REQ_FIRST_ITEM;
                case KEY_LL:        return  REQ_LAST_ITEM;

                case KEY_LEFT:      return  REQ_LEFT_ITEM;
```

continued

```
            case KEY_RIGHT:        return  REQ_RIGHT_ITEM;
            case KEY_UP:           return  REQ_UP_ITEM;
            case KEY_DOWN:         return  REQ_DOWN_ITEM;

            case 0x15:    /* ^U */return  REQ_SCR_ULINE;
            case 0x04:    /* ^D */return  REQ_SCR_DLINE;
            case 0x06:    /* ^F */return  REQ_SCR_DPAGE;
            case 0x02:    /* ^B */return  REQ_SCR_UPAGE;

            case 0x18:    /* ^X */return  REQ_CLEAR_PATTERN;
            case 0x08:    /* ^H */return  REQ_BACK_PATTERN;
            case 0x01:    /* ^A */return  REQ_NEXT_MATCH;
            case 0x1a:    /* ^Z */return  REQ_PREV_MATCH;

            case 0x14:    /* ^T */return  REQ_TOGGLE_ITEM;
        }
    return c;

}
```

Figure 10-22: Sample Routine that Translates Keys into Menu Requests

Note that because **wgetch()** here automatically does a refresh before reading a character, you can omit explicit calls to **wrefresh()** in applications that do character input.

ETI Menu Requests

ETI menu requests are made by calling function **menu_driver()** with an **int** value that signifies the request. To appreciate the effects of some requests, bear in mind what a menu page is.

> A menu page is the collection of currently visible menu items, i.e., those displayed in the menu subwindow.

A menu page is distinct from a form page, which is a logical portion of a

form. Form pages are treated in the upcoming section, "Forms."

Item Navigation Requests

These requests enable your end user to navigate from item to item whether or not the items are displayed at the moment.

REQ_NEXT_ITEM	- move to next item
REQ_PREV_ITEM	- move to previous item
REQ_FIRST_ITEM	- move to first item
REQ_LAST_ITEM	- move to last item

The order of the items in the array originally passed to **new_menu()** or **set_menu_items()** determines the order in which items are visited in response to these requests.

A REQ_NEXT_ITEM request from the last item or a REQ_PREV_ITEM request from the first item returns the value E_REQUEST_DENIED.

Often, a scrolling operation not explicitly requested by the user may nonetheless take place in response to these requests. For example, the REQ_FIRST_ITEM request on a menu that is not currently displaying the first item may scroll to display the menu's first item at the top of the screen.

Directional Item Navigation Requests

These requests enable your end user to navigate from item to item in different directions.

REQ_LEFT_ITEM	- move left to item
REQ_RIGHT_ITEM	- move right to item
REQ_UP_ITEM	- move up to item
REQ_DOWN_ITEM	- move down to item

Directional item navigation requests are not cyclic. If there is no item on the current page to the left or right of the current item, the menu driver returns E_REQUEST_DENIED in response to the corresponding request.

On the other hand, if the menu is scrollable and there are more items above or below the current menu page, the corresponding requests REQ_UP_ITEM and REQ_DOWN_ITEM generate an automatic scrolling operation. If not, the menu driver returns E_REQUEST_DENIED.

Menu Scrolling Requests

These requests enable your users to scroll easily through menus that span more than one menu page.

REQ_SCR_DLINE	- scroll menu down a line
REQ_SCR_ULINE	- scroll menu up a line
REQ_SCR_DPAGE	- scroll menu down a page
REQ_SCR_UPAGE	- scroll menu up a page

The current and top items are adjusted by these operations.

Menu scrolling requests are also not cyclic. Attempts to scroll up from the first menu page, or scroll down from the last, return from the menu driver the value E_REQUEST_DENIED.

Multi-Valued Menu Selection Request

This request enables your end user to select or deselect an item in a multi-valued menu.

REQ_TOGGLE_ITEM	- select/deselect item

If the item is currently selected, this request deselects it, and vice versa.

To use this request, the O_ONEVALUE option must be off. (See "Setting and Fetching Menu Options.") If the option is on, you have a single-valued menu. In that case, this request fails and E_REQUEST_DENIED is returned from the menu driver.

Pattern Buffer Requests

The pattern buffer is an area automatically allocated for your menu application programs. It is used to position the current menu item at an item name that matches the pattern. You can modify the pattern buffer

■ by calling **set_menu_pattern()** (described below)

■ by passing the menu driver printable ASCII characters one at a time

Each nonprintable ASCII character that is received by the menu driver is assumed to be a menu request. On the other hand, each printable ASCII character that is received by the menu driver is entered into the pattern buffer. At the same time, the current item advances to the first matching item. If no matching item is found, the current item remains unchanged, the character is deleted from the pattern buffer, and the menu driver returns E_NO_MATCH.

The following requests enable you to change and read the pattern buffer.

REQ_CLEAR_PATTERN	- clear pattern buffer
REQ_BACK_PATTERN	- delete last character from pattern buffer
REQ_NEXT_MATCH	- move to next pattern match
REQ_PREV_MATCH	- move to previous pattern match

Request REQ_CLEAR_PATTERN clears the pattern buffer entirely.
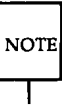
> **NOTE** Without request REQ_CLEAR_PATTERN, the pattern buffer is automatically cleared after each successful scrolling or item navigation operation. In other words, anytime the top item or current item changes, the pattern buffer is cleared automatically.

REQ_BACK_PATTERN deletes the last character from the pattern buffer. This request can be used to support a backspace operation on the pattern buffer.

Sometimes more than one menu item will match the character(s) entered by the user. REQ_NEXT_MATCH moves the user forward on the displayed menu to the next array item that matches the data in the pattern buffer. REQ_PREV_MATCH, on the other hand, moves the user backward on the displayed menu to the previous array item that matches the pattern buffer. In both cases, if no additional match is found, the current item remains unchanged and E_NO_MATCH is returned from the menu driver.

Requests REQ_NEXT_MATCH and REQ_PREV_MATCH are cyclic through all menu items. In addition, these requests generate automatic scrolling requests if the menu is scrollable and the next or previous matching item is not visible.

> **NOTE** An empty pattern buffer matches all items.

Application-Defined Commands

ETI menu requests are implemented as integers above the **curses** maximum key value KEY_MAX. A symbolic constant MAX_COMMAND is provided to enable your applications to implement their own requests (commands) without conflicting with the ETI form and menu system. All menu requests are greater than KEY_MAX and less than or equal to MAX_COMMAND. Your application-defined requests should be greater than MAX_COMMAND. Two illustrations are given in the following example. Figure 10-23 diagrams this relationship between ETI key values, ETI menu requests, and your application program's menu requests.
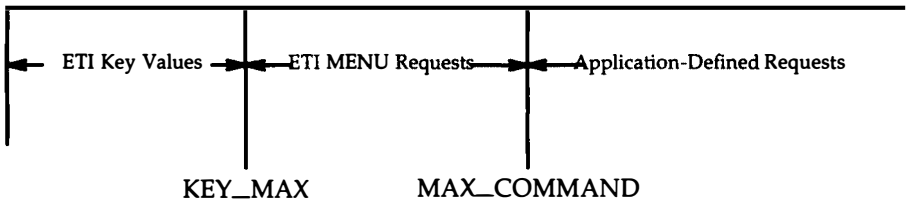
Figure 10-23: Integer Ranges for ETI Key Values and MENU Requests

Calling the Menu Driver

The menu driver checks whether the virtualized character passed to it is an ETI menu request. If so, it performs the request and reports the results. If the character is not a menu request, the menu driver checks if the character is data, i.e., a printable ASCII character. If so, it enters the character in the pattern buffer and looks for the first match among the item names. If no match is found, the menu driver deletes the character from the pattern buffer and returns E_NO_MATCH. If the character is not recognized as a menu request or data, the menu driver assumes the character is an application-defined command and returns E_UNKNOWN_COMMAND.

To illustrate a sample design for calling the menu driver, we will consider a program that permits interaction with a menu of astrological signs. Figure 10-24 displays the menu.

```
+------------------------------+
| Aries       The Ram          |
| Taurus      The Bull         |
| Gemini      The Twins        |
| Cancer      The Crab         |
| Leo         The Lion         |
| Virgo       The Virgin       |
| Libra       The Balance      |
| Scorpio     The Scorpion     |
| Sagittarius The Archer       |
| Capricorn   The Goat         |
| Aquarius    The Water Bearer |
| Pisces      The Fishes       |
+------------------------------+
```

Figure 10-24: Sample Menu Output (2)

You have already seen much of the astrological sign program in previous examples. Its function **get_request()**, for instance, appeared in Figure 10-22. Figure 10-25 shows its remaining routines.

```
        /* This program displays a sample menu.

        Omitted here are the key mapping defined by get_request()
        in Figure 10-22; application-defined routines display_menu()
        and erase_menu() in Figure 10-21; and the curses initialization
        routine start_curses in section, "ETI Low-Level Interface to
        High-Level Functions"   */

#include <string.h>
#include <menu.h>
```

continued

```
static char *    PGM   = (char *) 0; /* program name */


static int my_driver (m, c)    /* handle application commands */
MENU * m;
int c;
{
        switch (c)
        {
                case QUIT:
                        return TRUE;
                        break;
        }
        beep (); /* signal error */
        return FALSE;
}

main (argc, argv)
int argc;
char * argv[];
{
        WINDOW * w;
        MENU *        m;
        ITEM **       i;
        ITEM **       make_items ();
        void          free_items ();
        int           c, done = FALSE;

        PGM = argv[0];
        start_curses ();

        if (! (m = new_menu (make_items ())))
                error ("error return from new_menu", NULL);

        display_menu (m);

        /* interact with user */

        w = menu_win (m);

        while (! done)
        {
```

```
                switch (menu_driver (m, c = get_request (w)))
                {
                        case E_OK:
                                break;
                        case E_UNKNOWN_COMMAND:
                                done = my_driver (m, c);
                                break;
                        default:
                                beep ();/* signal error */
                                break;
                }
        }
        erase_menu (m);
        end_curses ();
        i = menu_items (m);
        free_menu (m);
        free_items (i);
        exit (0);
}

typedef struct
{
        char *          name;
        char *          desc;
}

        ITEM_RECORD;

        /* item definitions */

static ITEM_RECORD signs [] =
{
        "Aries",        "The Ram",
        "Taurus",       "The Bull",
        "Gemini",       "The Twins",
        "Cancer",       "The Crab",
        "Leo",          "The Lion",
        "Virgo",        "The Virgin",
        "Libra",        "The Balance",
        "Scorpio",      "The Scorpion",
        "Sagittarius",  "The Archer",
        "Capricorn",    "The Goat",
        "Aquarius",     "The Water Bearer",
        "Pisces",       "The Fishes",
```

```
                                                        continued
          (char *) 0,      (char *) 0,
};

#define MAX_ITEM  512

static ITEM *            items [MAX_ITEM + 1]; /* item buffer */

static ITEM ** make_items () /* create the items */
{
        int i;

        for (i = 0; i < MAX_ITEM && signs[i].name; ++i)
                items[i] = new_item (signs[i].name, signs[i].desc);

        items[i] = (ITEM *) 0;
        return items;
}

static void free_items (i) /* free the items */
ITEM ** i;
{
        while (*i)
                free_item (*i++);
}
```

Figure 10-25: Sample Program Calling the Menu Driver

——

Function **main()** first calls the application-defined routine **make_items()** to create the items from the array **signs**. The value returned is passed to **new_menu()** to create the menu. Function **main()** then initializes **curses** using **start_curses()** and displays the menu using **display_menu()**.

In its **while** loop, **main()** repeatedly calls **menu_driver()** with the character returned by **get_request()**. If the menu driver does not recognize the character as a request or data, it returns E_UNKNOWN_COMMAND, whereupon the application-defined routine **my_driver()** is called with the same character. Routine **my_driver()** processes the application-defined commands. In this example, there is only one, QUIT. If the character passed does not signify QUIT, **my_driver()** signals an error and returns FALSE—the signal prompts the user to re-enter the character. If the character passed is the QUIT character, **my_driver()** returns TRUE. In turn, this sets **done** to TRUE, and the **while** loop is exited.

Finally, **main()** erases the menu, terminates low-level ETI **(curses)**, frees the menu and its items, and exits the program.

This example shows a typical design for calling the menu driver, but it is only one of several ways you can structure a menu application. For another example, see the demonstration program **menu2.c** delivered with the ETI product.

If the **menu_driver()** recognizes and processes the input character argument, it returns E_OK. In the following error situations, the **menu_driver()** returns the indicated value:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null menu
E_BAD_STATE	- called from init/term routines
E_NOT_POSTED	- menu is not posted
E_UNKNOWN_COMMAND	- unknown command
E_NO_MATCH	- item match failed
E_REQUEST_DENIED	- recognized request failed

NOTE

Because the menu driver calls the initialization and termination routines described in the next section, it may not be called from within them. Any attempt to do so returns E_BAD_STATE.

Establishing Item and Menu Initialization and Termination Routines

Sometimes, you may want the menu driver to execute a specific routine during the change of an item or menu. The following functions let you do this easily.

SYNOPSIS

```
typedef void (*PTF_void) ();

int set_menu_init (menu, func)
MENU * menu;
PTF_void func;

PTF_void menu_init (menu)
MENU * menu;

int set_menu_term (menu, func)
MENU * menu;
PTF_void func;

PTF_void menu_term (menu)
MENU * menu;

int set_item_init (menu, func)
MENU * menu;
PTF_void func;

PTF_void item_init (menu)
MENU * menu;

int set_item_term (menu, func)
MENU * menu;
PTF_void func;

PTF_void item_term (menu)
MENU * menu;
```

The argument **func** is a pointer to the specific function you want executed by the menu driver. This application-defined function takes a menu pointer as an argument.

If you want your application to execute an application-defined function at one of the initialization or termination points listed below, you should call the appropriate **set_** routine at the start of your program. If you do not want a specific function executed in these cases, you may refrain from calling these routines altogether.

The following paragraphs summarize when each initialization and termination routine is executed.

Function set_menu_init()

The argument **func** to this function is automatically called by the menu system:

- Just before the menu is posted

- just after each menu scrolling operation, i.e., every time the top row changes on a posted menu, whether by the menu driver in response to a request or by a program's call to **set_current_item()** or **top_row()**

Function set_item_init()

The argument **func** is automatically called by the menu system:

- just before the menu is posted

- just after the current item on a posted menu is changed, whether by the menu driver's response to a request or by a program's call to **set_current_item()** or **top_row()**

Function set_item_term()

The argument **func** is automatically called by the menu system:

- just before the current item changes on a posted menu

- just before the menu is unposted

Function set_menu_term()

The argument **func** is automatically called by the menu system:

■ just before a scrolling operation on a posted menu

■ just before the menu is unposted

If functions **set_menu_init()**, **set_menu_term()**, **set_item_init()**, or **set_item_term()** encounter an error, they return

    E_SYSTEM_ERROR   - system error

Figure 10-26 shows how you can use function **set_item_init()** to implement a menu prompting feature as your end user moves from item to item.

```
WINDOW * prompt_window;

void display_prompt (s)
char * s;
{
        WINDOW * w = prompt_window;

        werase (w);
        wmove (w, 0, 0);           /* move to window origin */
        waddstr (w, s);            /* write prompt in window */
        wrefresh (w);              /* display prompt */
}
void generate_prompt (m)
MENU * m;
{
        /* display the prompt string associated with the current item */

        char * s = item_userptr (current_item (m));
        display_prompt (s);
}
ITEM * items[NUMBER_OF_ITEMS + 1];

main ()
{
        MENU * m;

        for (i = 0; i < NUMBER_OF_ITEMS; ++i)
        {
                /* read in name and prompt strings here */

                items[i] = new_item (name, "");
                set_item_userptr (items[i], prompt);
        }
        items[i] = (ITEM *) 0;

        m = new_menu (items);
        set_item_init (m, generate_prompt);  /* set initialization routine */
}
```

Figure 10-26: Using an Initialization Routine to Generate Item Prompts

Function **set_item_init()** arranges to call **generate_prompt()** whenever the

menu item changes. Function **generate_prompt()** fetches the item user pointer associated with the current item and calls **display_prompt()**, which displays the item prompt. Function **display_prompt()** is a separate function to enable you to use it for other prompts as well.

Fetching and Changing the Current Item

The current item is the item where your end user is positioned on the screen. Unless it is invisible, this item is highlighted and the cursor rests on the item. To have your application program set or determine the current item, use the following functions.

    SYNOPSIS

    int set_current_item (menu, item)
    MENU * menu;
    ITEM * item;

    ITEM * current_item (menu)
    MENU * menu;

    int item_index (item)
    ITEM * item;

Function **set_current_item()** enables you to set the current item by passing an item pointer, while function **current_item()** returns the pointer to the current item.

The function **item_index()** takes an item pointer argument and returns the index to that item in the item pointer array. The value of this index ranges from 0 through N-1, where N is the total number of items connected to the menu.

Because the menu driver satisfies ETI-defined item navigation requests automatically, your application program need not call **set_current_item()**, unless you want to implement additional item navigation requests for your application. You may, for instance, want a request to jump to a particular item or an item, say, two items down from the current one on the menu page.

When a menu is created by **new_menu()** or the items associated with a menu are changed by **set_menu_items**, the current item is set to the first item of the menu.

As an example of **set_current_item()**, the following function sets the current item of menu **m** to the first item of the menu:

```
int set_first_item (m) /* set current item to first item */
MENU * m;
{
                ITEM ** i = menu_items (m);
                return set_current_item (m, i[0]);
}
```

As an example of **current_item()**, the following routine checks if the first menu item is the current one:

```
int first_item (m) /* check if current item is first item */
MENU * m;
{
                ITEM * i = current_item (m);
                return item_index (i) == 0;
}
```

If successful, function **set_current_item()** returns E_OK. If an error occurs, function **set_current_item()** returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null menu pointer or item not connected to menu
E_BAD_STATE	- called from initialization or termination routines

Function **current_item()** returns (ITEM *) 0 if given a NULL menu pointer or there are no items connected to the menu.

Function **item_index()** returns -1 if the item pointer is NULL or the item is not connected to a menu.

Fetching and Changing the Top Row

Function **top_row()** returns the number of the menu row currently displayed at the top of your end user's menu. Function **set_top_row()** sets the top of the menu to the named row, unless the row does not start a complete page of items. In this case, it returns E_BAD_ARGUMENT.

```
int set_top_row(menu, row)
MENU * menu;
int row;

int top_row(menu)
MENU * menu;
```

Function **set_top_row()** sets the current item to the leftmost item in the new top row. Variable **row** must be in the range of 0 through TR-VR, where TR is the total number of rows as determined by the menu format and VR is the number of visible rows. If the value of **row** is greater, the row does not start a complete page of items. See "Specifying the Menu Format" for details on menu display.

When a menu is created by **new_menu()** or the items associated with the menu are changed by **set_menu_items**, the top row is set to 0.

| NOTE | If the menu format or the O_ROWMAJOR option is changed, the top row is automatically set to 0. See "Specifying the Menu Format" and "Setting and Fetching Menu Options" for details on changing these menu attributes. |

In addition, if the current item is changed by **set_current_item()** or **set_menu_pattern()** to an item that is not currently visible, the top row is generally set to the row that contains the new current item. The sole exception occurs when, as noted above, the top row does not start a complete page of items.

If successful, function **set_top_row()** returns E_OK. If an error occurs, **set_top_item()** returns one of the following error messages:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- NULL menu pointer or index out of range
E_BAD_STATE	- called from init/term routines
E_NOT_CONNECTED	- no connected items

Function **top_row()** returns -1 if given a NULL menu pointer or no items are connected to the menu.

Positioning the Menu Cursor

Some applications may need to move the menu's window cursor from the position required for continued processing by the ETI menu driver. To move the cursor back to where it belongs, you use function **pos_menu_cursor()**.

```
SYNOPSIS

int pos_menu_cursor (menu)
MENU * menu;
```

If your application does not change the cursor position in the menu window, it will not be necessary to call this function.

Your application might change the cursor position automatically as the result of prior calls to menu driver initialization routines such as **set_item_init()**. Or it might do so as the result of explicit calls to application routines such as writing a prompt. Figure 10-27 illustrates this usage.

```
void generate_prompt (m)
MENU * m;
{

        /* display the prompt string associated with the current item */

        WINDOW * w = menu_win (m);
        char * s = item_userptr (current_item (m));
        box (w, 0, 0);
        wmove (w, 0, 0);
        waddstr (w, s);
        pos_menu_cursor (m);
}
```

Figure 10-27: Returning Cursor to its Correct Position for Menu Driver Processing

---

If function **pos_menu_cursor()** is successful, it returns E_OK. In the following error situations, it fails and returns the indicated value:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null menu pointer
E_NOT_POSTED	- menu is not posted

Changing and Fetching the Pattern Buffer

Remember that the pattern buffer is used to make the first item that matches the pattern the current item. In general, to match the current menu item, your application program inserts characters into the pattern buffer that have been passed to the menu driver from the user's data entry. As an alternative, you can insert characters into the pattern buffer with the function **set_menu_pattern()**.

SYNOPSIS

```
int set_menu_pattern (menu, pattern)
MENU * menu;
char * pattern;


char * menu_pattern (menu)
MENU * menu;
```

Function **set_menu_pattern()** first clears the pattern buffer and then adds the characters in **pattern** to the buffer until **pattern** is exhausted. The function next tries to find the first item that matches the **pattern**. If it does not find a complete match, the pattern buffer is cleared and the current item does not change. If **pattern** is the null string (" "), the pattern buffer is simply cleared. The pattern buffer is automatically cleared in the following situations:

■ Each successful scrolling or item navigation operation is completed (in other words, whenever the top or current item changes).

■ A menu is created by **new_menu()**.

■ The items associated with a menu are changed by **set_menu_items**.

If successful, function **set_menu_pattern()** returns E_OK. If an error occurs, function **set_menu_pattern()** returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- NULL menu pointer or NULL pattern pointer
E_NO_MATCH	- complete match failed

Function **menu_pattern()** returns the value of the string in the pattern buffer. If the pattern buffer is empty (the null string " "), it returns the null string (" "). If the menu pointer argument is NULL, it returns NULL, i.e., (char *) 0.

To determine if your user has entered data that matches an item, you might write a routine that uses **set_menu_pattern()**, as follows.

```
int find_match (m, newpattern)   /* returns TRUE or FALSE */
MENU * m;
char * newpattern;
{
        return set_menu_pattern(m, newpattern) == E_OK;
}
```

If the **newpattern** matches a menu item, function **set_menu_pattern()** returns E_OK and hence **find_match()** returns TRUE. In addition, **find_match()** advances the current item to the matching item.

Manipulating the Menu User Pointer

As it does for panels and forms, ETI provides user pointers for each menu. You can use these pointers to reference menu messages, titles, and the like.

```
SYNOPSIS

int set_menu_userptr (menu, userptr)
MENU * menu;
char * userptr;

char * menu_userptr (menu)
MENU * menu;
```

By default, the menu user pointer (what **menu_userptr()** returns) is NULL.

If successful, **set_menu_userptr()** returns E_OK. If an error occurs, it returns the following:

E_SYSTEM_ERROR   - system error

The code in Figure 10-28 illustrates how you can use these two functions to display a title for your menu. Function **main()** sets the menu user pointer to point to the title of the menu. Later, function **display_menu()** initializes the title with the value returned by **menu_userptr()**. We have previously seen a version of **display_menu()** in Figure 10-25.

```
static void display_menu (m)      /* create menu windows and post */
MENU * m;
{
        char *            title = menu_userptr (m);   /* fetch menu title */

        WINDOW * w;
        int               rows;
        int               cols;

        scale_menu (m, &rows, &cols);   /* get dimensions of menu */

        /* create menu window and subwindow */

        if (w = newwin (rows+2, cols+2, 0, 0))
        {
                set_menu_win (m, w);
                set_menu_sub (m, derwin (w, rows, cols, 1, 1));
                box (w, 0, 0);
                keypad (w, 1);
        }
        else
                error ("error return from newwin", NULL);

        if (post_menu (m) != E_OK)
                error ("error return from post_menu", NULL);
        if (title)                      /* if title set */
        {
                size = strlen (title);
                wmove (w, 0, (cols-size)/2+1);   /* position cursor */
                waddstr (w, title);   /* write title */
        }
}
main ()
{
        MENU * m;
        char * menutitle;      /* initialize menutitle to desired string */

        set_menu_userptr (m, menutitle); /* set user pointer to point to title */
        display_menu (m);
}
```

Figure 10-28: Example Setting and Using A Menu User Pointer

If function **set_menu_userptr()** is passed a NULL menu pointer, like all ETI functions, it assigns a new current default menu user pointer. In the following, the new default is the string "Default Menu Title."

```
MENU * m;

char * userprtr = "Default Menu Title";

set_menu_userptr( (MENU *) 0, userptr);
/* sets new default userptr */
```

Setting and Fetching Menu Options

ETI provides several menu options, some of which we have already discussed. Two functions manipulate options: one sets them, the other returns their settings.

```
SYNOPSIS

int set_menu_opts (menu, opts)
MENU * menu;
OPTIONS opts;

OPTIONS menu_opts (menu)
MENU * menu;

options:
                    O_ONEVALUE
                    O_SHOWDESC
                    O_ROWMAJOR
                    O_IGNORECASE
                    O_SHOWMATCH
                    O_NONCYCLIC
```

Besides turning the named options on, function **set_menu_opts()** turns off all other menu options. By default, all menu options are on.

The menu options and their effects are as follows:

O_ONEVALUE   determines whether the menu is a single-valued or multi-valued. In general, menus are single-valued and this option is on. Recall that upon exit from single-valued menus, your application queries the current item to ascertain the item selected. Turning off this option signifies a multi-valued menu. One way to select several items is to use the REQ_TOGGLE_ITEM request, another is to call **set_item_value()**. (See the previous section, "Manipulating an Item's Select Value in a Multi-Valued Menu.") Recall that your application must examine each item's select value to determine whether it has been selected. When this option is on, all item select values are FALSE.

O_SHOWDESC   determines whether or not the description of an item is displayed. By default, this option is on and both the item name and description are displayed. If this option is off, only the name is displayed.

O_ROWMAJOR   determines how the menu items are presented on the screen—in row-major or column-major order. In row-major order, menu items are first displayed left to right, then top to bottom. In column-major order, they are displayed first top to bottom, then left to right. By default, this option is on, so menu items are displayed in row-major order. If the option is off, the items are displayed in column-major order. See "Specifying the Menu Format" for more details on how menus are displayed.

O_IGNORECASE   instructs the menu driver to ignore upper- and lower-case during the item match operation. If this option is off, character case is not ignored and the match must be exact.

O_SHOWMATCH   determines whether visual feedback is provided as each item's data entry is processed. Ordinarily, as soon as a match occurs, the cursor is advanced through the item to reflect the contents of the pattern buffer. If this option is off, however, the cursor remains to the left of the current item.

O_NONCYCLIC   determines how REQ_NEXT_ITEM and REQ_PREV_ITEM behave when the current item is the last or first item. Default is to not allow cycling from first to last item or from last to first item. If O_NONCYCLIC is turned off, REQ_NEXT_ITEM from the last item causes the first item to become the current. If O_NONCYCLIC is turned off, REQ_PREV_ITEM from the first item causes the last item to become current.

Like all ETI options, menu OPTIONS are Boolean values, so you use Boolean operators to turn them on or off with functions **set_menu_opts()** and **menu_opts()**. For example, to turn off option O_SHOWDESC for menu **m0** and turn on the same option for menu **m1**, you can write:

```
MENU * m0, * m1;

set_menu_opts (m0, menu_opts (m0) & ~O_SHOWDESC); /* turn option off */
set_menu_opts (m1, menu_opts (m1) |  O_SHOWDESC); /* turn option on  */
```

ETI provides two alternative functions for turning options on and off for a given menu.

```
SYNOPSIS

int menu_opts_on (menu, opts)
MENU * menu;
OPTIONS opts;

int menu_opts_off (menu, opts)
MENU * menu;
OPTIONS opts;
```

Unlike function **set‑menu‑opts()**, these functions do not affect options that are unmentioned in their second argument. In addition, if you want to change one option, you need not apply Boolean operators or use **menu‑opts()**.

For example, the following code turns option O‑SHOWDESC off for menu **m0** and on for menu **m1**:

```
MENU * m0, * m1;

menu_opts_off (m0, O_SHOWDESC); /* turn option off */
menu_opts_on  (m1, O_SHOWDESC); /* turn option on  */
```

As usual, you can change the current default for each option by passing a NULL menu pointer. For instance, to turn the default option O‑SHOWDESC off, you write

```
menu_opts_off ((MENU *) 0, O_SHOWDESC); /* turn default option off */
```

In general, functions **set_menu_opts()**, **menu_opts_on()**, and **menu_opts_off()** return E_OK. If an error occurs, they return one of the following:

      E_SYSTEM_ERROR   - system error
      E_POSTED   - menu is posted

Forms

A form is a collection of one or more pages of fields. The fields may be used for titles, labels to guide the user, or for data entry. Figure 10-29 displays a simple form with five fields including two for data entry.

Figure 10-29: Sample Form Display

Compiling and Linking Form Programs

To use the form routines, you specify

#include <form.h>

in your C program files and compile and link with the command line

cc [*flags*] *files* **-lform -lcurses** [*libraries*]

If you want to use the menu or panel routines as well, place the appropriate **-1** option before the option **-lcurses.**

Overview: Writing Form Programs in ETI

This section introduces the basic ETI form terminology, lists the steps in a typical form application, and reviews the sample program that produced the output as shown in Figure 10-29.

Some Important Form Terminology

The following terms are helpful in working with ETI form functions:

field	an m x n block of character positions within a form that ETI functions can manipulate as a unit
active field	a field that is visited during form processing for data entry, change, selection, and so forth
inactive field	a field that is completely ignored during form processing, such as a title, field marker or other label
form	a collection of one or more pages of fields
connecting fields to a form	associating an array of field pointers with a form
page	a logical subdivision of a form usually occupying one screen
posting a form	writing a form on its associated subwindow
unposting a form	erasing a form from its associated subwindow
freeing a form	deallocating the memory for a form and, as a by-product, disconnecting the previously associated array of field pointers from the form
freeing a field	deallocating the memory for a field

NULL   generic term for a null pointer cast to
                                                  the type of the particular object—
                                                  field, form, and so on

What a Typical Form Application Program Does

In general, a form application program will

■ initialize low-level ETI (**curses**)

■ create the fields for the form

■ create the form

■ post the form

■ refresh the screen

■ process end user form requests

■ unpost the form

■ free the form

■ free the fields

■ terminate low-level ETI (**curses**)

A Sample Form Application Program

Figure 10-30 shows the ETI program necessary for producing the form in
Figure 10-29.

```
#include <form.h>
#include <string.h>

FIELD * make_label (frow, fcol, label)
int frow;        /* first row      */
int fcol;        /* first column   */
char * label;    /* label          */
```

continued

```
{
        FIELD * f = new_field (1, strlen (label), frow, fcol, 0, 0);

        if (f)
        {
                set_field_buffer (f, 0, label);
                set_field_opts (f, field_opts(f) & ~O_ACTIVE);
        }
        return f;
}

FIELD * make_field (frow, fcol, cols)
int frow;/* first row          */
int fcol;/* first column        */
int cols;/* number of columns   */
{
        FIELD * f = new_field (1, cols, frow, fcol, 0, 0);

        if (f)
                set_field_back (f, A_UNDERLINE);
        return f;
}

main ()
{
        FORM *          form;
        FIELD *         f[6];
        int             i = 0;
/*
        ETI initialization
*/
        initscr ();
        nonl ();
        raw ();
        noecho ();
        wclear (stdscr);
/*
        create fields
*/
        f[0] = make_label (0, 7, "Sample Form");
        f[1] = make_label (2, 0, "Field 1:");
        f[2] = make_field (2, 9, 16);
        f[3] = make_label (3, 0, "Field 2:");
```

```
        f[4] = make_field (3, 9, 16);
        f[5] = (FIELD *) 0;
/*
        create and display form
*/
        form = new_form (f);
        post_form (form);
        wrefresh (stdscr);
        sleep (5);
/*
        erase form and free both form and fields
*/
        unpost_form (form);
        wrefresh (stdscr);
        free_form (form);

        while (f[i])
                free_field (f[i++]);
/*
        ETI termination
*/
        endwin ();
        exit (0);
}
```

Figure 10-30: Code To Produce a Simple Form

In this example, all text within the form is associated with a field. Fields may be active or inactive: active fields are affected by form processing, inactive fields are not. The underlined fields are active, whereas the label fields, "Sample Form", "Field 1:", and "Field 2:", are inactive.

Turn now to the program itself. This example starts with two **#include** files. Every form program must include the header file **form.h**, which contains important definitions of form objects. This particular program uses the C string library function **strlen()**, so it includes the header file **string.h**, whose definitions the string library function needs. See **string**(3C) in the *Programmer's Reference Manual* for details.

Next, there are two programmer-defined functions **make_label()** and **make_field()**, which we will discuss in a moment. Consider procedure **main()**. It declares three objects:

- **form**, a pointer to a form

- **f[6]**, an array of field pointers

- **i**, an index variable, initialized to 0

The first five functions initialize low-level ETI **(curses)** for high-level ETI form functions. Function **initscr()** initializes the screen, **nonl()** ensures that a carriage return on using **wgetch()** will not automatically generate a newline, **raw()** passes input characters uninterpreted to your program, **noecho()** disables echoing of your user's input (the form functions provide echoing where appropriate), and **wclear(stdscr)** clears the standard screen.

The statements that create the form's fields and labels in this example make calls to the programmer-defined functions **make_label()** and **make_field()**. You can do without these programmer-defined functions, but you may find them convenient. Both of them use the ETI function **new_field()**. They take three arguments, which correspond to three of the six arguments of **new_field()**.

The first argument of **new_field()** is the number of rows of the field. In this example, it is always one. The last two arguments are often 0 as they are here; they will be explained in the next section. The second argument of **new_field()** is the number of columns in the field. This number is determined from the third parameter in **main()**'s calls to **make_label()** and **make_field()**. For the label fields, the calls to **make_label()** pass the string that is to constitute the field so that **strlen()** can be used to count the length or number of columns of the string. For the fields to be edited by the end-user (had this example permitted entering data into the fields), calls to **make_field()** simply pass the number of columns directly.

The third and fourth arguments to **new_field()** correspond to the first and second arguments to **make_label()** and **make_field()**. They are the starting position (**firstrow, firstcol**) of the label or field in the form subwindow. (In this example, the default subwindow **stdscr** is used.) The last assignment to **f[5]** terminates the array with the NULL field pointer.

Once the function **make_label()** creates the field for the label, it places the label in the field using function **set_field_buffer()**. The second argument to this function is 0 because the value of a field is stored in buffer 0. Finally, function **make_label()** calls **set_field_opts()**, which turns off the O_ACTIVE option for the field. This means that the field is ignored during form driver processing.

On the other hand, once the function **make_field()** creates the field proper, it sets the field's background attribute to A_UNDERLINE. This has the effect of underlining the field so that it is visible.

After you create the fields for a form, you create the form itself using **new_form()**. This function takes the pointer to the array of field pointers and connects the fields to the form. The pointer returned is stored in variable **form**—it will be passed to subsequent form manipulation routines. To display the form, function **post_form()** posts it on the default subwindow **stdscr**, while **wrefresh(stdscr)** actually displays this subwindow on the terminal screen. The display remains for 5 seconds, as determined by **sleep()**.

At this point, most forms would accept and process user input. To illustrate a very simple form, this program does not accept user input.

To erase the form, you first unpost it using **unpost_form()**. This erases it from the form subwindow. The call to **wrefresh()** actually erases the form from the display screen. Function **free_form()** disconnects the form from its array of field pointers **f**.

The **while()** loop, starting with the first field in the field pointer array, frees each field referenced in the array. The effect is to deallocate the space for each field.

We have met the last two lines of the program before. Function **endwin()** terminates low-level ETI, while **exit(0)** terminates the program.

There are many ETI form routines not listed in Figure 10-30. These enable you to tailor your form programs to suit local needs and preferences. The following sections explain how to use all ETI form routines. Each routine is illustrated with one or more code fragments. Many of these are drawn from two larger form application programs listed at the end of the chapter. By

reviewing the code fragments, you will come to understand the larger programs.

Creating and Freeing Fields

To create a form, you must first create its fields. The following functions enable you to create fields and later free them.

```
SYNOPSIS

FIELD * new_field (rows, cols, firstrow, firstcol, nrow, nbuf)
int rows, cols, firstrow, firstcol, nrow, nbuf;

FIELD * dup_field (field, firstrow, firstcol)
FIELD * field;
int firstrow, firstcol;

FIELD * link_field (field, firstrow, firstcol)
FIELD * field;
int firstrow, firstcol;

int free_field (field)
FIELD * field;
```

Unlike menu items which always occupy one row, the fields on a form may contain one or more rows. Function **new_field()** creates and initializes a new field that is **rows** by **cols** large and starts at point (**firstrow, firstcol**) relative to the origin of the form subwindow. All current system defaults are assigned to the new field when it is created using **new_field()**.

Variable **nrow** is the number of offscreen rows allocated for this field. Offscreen rows enable your program to display only part of a field at a given moment and let the user scroll through the rest. A zero value means that the entire field is always displayed, while a nonzero value means that the field is scrollable.

Variable **nbuf** is the number of additional buffers allocated for this field. You can use them to support default field values, undo operations, or other similar operations requiring one or more auxiliary field buffers.

Variables **rows** and **cols** must be greater than zero, while **firstrow, firstcol**, **nrow**, and **nbuf** must be greater than or equal to zero.

Each field buffer is ((**rows** + **nrow**) * **cols** + 1) characters large. (The extra character position holds the NULL terminator.) All fields have one buffer (namely, field buffer 0) that maintains the field's value. This buffer reflects any changes your end-user may make to the field. See the section, "Setting and Reading Field Buffers," for more details.

To create a form field **occupation** one row high and 32 columns wide, starting at position **2,15** in the form subwindow, with no offscreen rows and no additional buffers, you can write:

```
FIELD * occupation;

occupation = new_field (1, 32, 2, 15, 0, 0);  /* create field */
```

Generally you create all the fields for a form at the same point in your program, as Figure 10-30 demonstrated.

The function **dup_field()** duplicates an existing field at the new location **firstrow, firstcol**. During initialization, **dup_field()** copies nearly all the attributes of its field argument as well as its size and buffering information. However, certain attributes, such as being the first field on a page or having the field status set, are not duplicated in the newly created field. See the sections below, "Creating and Freeing Forms," and "Setting and Reading the Field Status," for details on these attributes.

Like **dup_field()**, function **link_field()** duplicates an existing field at a new location on the same form or another one. Unlike **dup_field()**, however, **link_field()** arranges that the two fields share the space allocated for the field buffers. All changes to the buffers of one field appear also in the buffers of the other. Besides enabling your user to enter data into two or more fields at once, this function is useful for propagating field values to later pages where only the first field is active (currently open to form processing). In this case, the inactive fields in effect become dynamic labels. See the section below, "Manipulating Field Options".

> NOTE: Linked fields share only the space allocated for the field buffers-the attribute values of either field may be changed without affecting the other.

Consider field **occupation** in the previous example. To duplicate it at location **3,15** and link it at location **4,15**, you write

```
FIELD * dup_occ, * link_occ;

dup_occ = dup_field (occupation, 3, 15);
link_occ = link_field (occupation, 4, 15);
```

Functions **new_field()**, **dup_field()**, and **link_field()** return a NULL pointer if there is no available memory for the FIELD structure or if they detect an invalid parameter.

Function **free_field()** frees all space allocated for the given field. Its argument is a pointer previously obtained from **new_field()**, **dup_field()**, or **link_field()**.

NOTE	To free a field, be sure that the field is not connected to a form.

As described in the section below, "Creating and Freeing Forms," you can disconnect fields from forms by using functions **free_form()** or **set_form_fields()**.

To free a form and all its fields, you write

```
FORM * form;

/* get pointer to form's field pointer array using form_fields() described in
   section below, "Changing the Fields on an Existing Form" */

FIELD ** f = form_fields (form);

free_form (form);        /* free form */

while (*f)
        free_field (*f++); /* free each field and increment pointer */
```

Notice that you free the form before its fields.

If successful, function **free_field()** returns E_OK. If not, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null field pointer
E_CONNECTED	- connected field

Remember that the field pointer returned by **new_field()**, **dup_field()**, or **link_field()** is passed to all field routines that record or examine the field's attributes. As with menu items, once a form field is freed, it must not be used again. Because the freed field pointer does not point to a genuine field, undefined results occur.

Manipulating Field Attributes

Recall that an attribute is any feature whose value can be set or read by an appropriate ETI function. A field attribute is a feature of a field whose value can be set or read by an appropriate ETI function. Field attributes include the field size and location.

Obtaining Field Size and Location Information

This function enables you to determine the defining characteristics of a field—its size, position, number of offscreen rows, and number of associated buffers.

SYNOPSIS

```
int field_info (field, rows, cols, firstrow, firstcol, nrow, nbuf)
FIELD * field;
int * rows, * cols, * firstrow, * firstcol, * nrow, * nbuf;
```

Because function **field_info()** must return more than a single value and C passes arguments to functions by "call by value" only, **field_info()** uses the pointer arguments **rows**, **cols**, **firstrow**, **firstcol**, **nrow**, and **nbuf**. These arguments are pointers to the locations used to return the requested information: the number of rows and columns comprising the field, the field starting location relative to the origin of its form subwindow, the number of offscreen rows, and the number of additional buffers.

As an example, consider how you might use **field_info()** to determine a field's buffer size. You fetch the field's number of onscreen and offscreen rows and number of columns, and do the arithmetic, thus:

```
int bufsize (f)
FIELD * f;
{
        int rows, cols, firstrow, firstcol, offrow, nbuf;

        field_info (f, &rows, &cols, &firstrow, &firstcol, &offrow, &nbuf);

        /* add up size of field and its terminator */

        return (rows + offrow) * cols + 1;
}
```

Note the use of the address operator **&** to pass **field_info()** the requisite pointers to the locations used to return the requested field information.

If successful, function **field_info()** returns E_OK. If not, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null field pointer

Moving a Field

ETI provides the following function to move an existing disconnected field to a new location.

SYNOPSIS

```
int move_field (field, firstrow, firstcol)
FIELD * field;
int firstrow;
int firstcol;
```

Figure 10-31 shows one way you might use function **move_field()**. Function **shift_fields()** receives the **int** value **updown**, which it uses to change the row number of each field in a given field pointer array. You could, of course, shift the columns in like fashion.

```
void shift_fields (f, updown)
FIELD ** f;
int updown; /* signed number of rows to shift */
{
        int rows, cols, frow, fcol, nrow, nbuf;

        while (*f)
        {

        /* field_info() fetches the values of the field parameters */

                field_info (*f, &rows, &cols, &frow, &fcol, &nrow, &nbuf);
                move_field (*f, frow + updown, fcol);
                ++f;
        }
}
```

Figure 10-31: Example Shifting All Form Fields a Given Number of Rows

See the previous section, "Obtaining Field Size and Location Information",
for more on **field_info()** used in this example.

If successful, function **move_field()** returns E_OK. If not, it returns one
of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null field or firstrow/firstcol < 0
E_CONNECTED	- connected field

Changing the Current Default Values for Field Attributes

ETI establishes initial current default values for field attributes. During
field initialization, every field attribute is assigned the current default value for
the attribute. As you can with menu functions, you can change or retrieve the
current default attribute values by calling the appropriate function with a

NULL field pointer. After the current default changes, every field created using **new_field()** will have the new default value.

NOTE	Fields previously created do not have their attributes changed by changing the current system default.

Several of the following sections show how to change the default values for various field attributes.

Setting the Field Type To Ensure Validation

Every field is created with the current default field type. The initial ETI default field type is a no_validation field. Any data may occupy it. (This default can be changed as described below.) To change a field's type from the default, ETI provides the following functions for manipulating a field's (data) type.

```
SYNOPSIS

int set_field_type (field, type, [arg_1, arg_2, ...])
FIELD * field;
FIELDTYPE * type;

FIELDTYPE * field_type (field)
FIELD * field;

char * field_arg (field)
FIELD * field;
```

The function **set_field_type()** takes a FIELDTYPE pointer and a variable number of arguments depending on the field type. The field type ensures that the field is validated as your end-user enters characters into the field or attempts to leave it.

The form driver (described later in the section, "Form Driver Processing") validates the data in a field only when data is entered by your end-user. Validation does NOT occur when

- the application program changes the field value by calling
  set_field_buffer()

- linked field values are changed indirectly—by changing the field to
  which they are linked

In all cases, validation occurs only if data is changed by passing data or mak-
ing requests to the form driver. To make requests, your user enters characters
or escape sequences mapped to commands that the form driver recognizes.
See the section below, "Form Driver Processing".

If successful, **set_field_type()** returns E_OK. If not, it returns the follow-
ing:

> E_SYSTEM_ERROR   - system error

Function **field_type()** returns the field type of the field, while function
field_arg() returns the field argument pointer. For more on the field argu-
ment pointer in programmer-defined field types, see the section below, "Sup-
porting Programmer-Defined Field Types."

If the function **set_field_type()** is applied to a NULL field, the field **type**
becomes the new current default.

> | NOTE | Remember that the initial ETI default is not to validate the field at all;
> any kind of data may be entered into the field.

You can change the ETI default by giving function **set_field_type()** a
NULL field pointer. Suppose, for instance, that you want to change the sys-
tem default field type to a minimum 10-character field of type
TYPE_ALNUM. As described below, this field type accepts alphanumeric
data—every entered character must be a digit or an alphabetic (not a special)
character. You can write

```
set_field_type ((FIELD *) 0, TYPE_ALNUM, 10);
```

ETI provides several generic field types besides TYPE_ALNUM. More-
over, you can define your own field types, as described later in the section,
"Creating and Manipulating Programmer-Defined Field Types." The follow-
ing sections describe all ETI generic field types.

TYPE_ALPHA

The form driver restricts a field of this type to alphabetic data.

```
SYNOPSIS

set_field_type (field, TYPE_ALPHA, width);
int width; /* minimum token width */
```

TYPE_ALPHA takes one additional argument, the minimum width specification of the field. Note that when you previously create a field with function **new_field()**, your **cols** argument is the maximum width specification of the field. With TYPE_ALPHA (and TYPE_ALNUM as well), your specification **width** must be less than or equal to **cols**. If not, the form driver cannot validate the field.

> | NOTE | TYPE_ALPHA does not allow blanks or other special characters.

To set a middlename field, for instance, to TYPE_ALPHA with a minimum of 0 characters (in effect, to make the end-user's completing the field optional), you can write

```
FIELD * middlename;

set_field_type (middlename, TYPE_ALPHA, 0);
```

TYPE_ALNUM

This type restricts the set field to alphanumeric data, alphabetic characters (upper- or lower-case) and digits.

```
SYNOPSIS

set_field_type (field, TYPE_ALNUM, width);
int width /* minimum token width */
```

Like TYPE_ALPHA, TYPE_ALNUM takes one additional argument, the field's minimum width specification.

> NOTE Like TYPE_ALPHA, TYPE_ALNUM does not allow blanks or other special characters.

To set a field, say **partnumber**, to receive alphanumeric data at least 8 characters wide, you write

```
FIELD * partnumber;

set_field_type (partnumber, TYPE_ALNUM, 8);
```

TYPE_ENUM

This field type enables you to restrict the valid data for a field to a set of enumerated values. The type takes three arguments beyond the minimum two that **set_field_type()** requires.

```
SYNOPSIS

set_field_type (field, TYPE_ENUM, keyword_list, checkcase, checkunique);
char ** keyword_list; /* list of acceptable values */
int checkcase;        /* check character case   */
int checkunique;      /* check for unique match */
```

The argument **keyword_list** is a NULL-terminated array of pointers to character strings that are the acceptable enumeration values. Argument **checkcase** is a Boolean flag that indicates whether upper- or lower-case is significant during match operations. Finally, **checkunique** is a Boolean flag indicating whether a unique match is required. If it is off and your end-user enters only part of an acceptable value, the validation procedure completes the field value automatically with the first matching value in the type. If it is on, the validation procedure completes the field value automatically only when enough characters have been entered to make a unique match.

To create a field, say **response**, with valid responses of "yes" ("y") or "no" ("n") in upper- or lower-case, you write the following:

```
char * yesno[] = { "yes", "no", (char *)0 };
FIELD * response;

set_field_type (response, TYPE_ENUM, yesno, FALSE, FALSE);
```

For an example that sets the last field (**checkunique**) to TRUE, see Figure 10-32, which sets the TYPE_ENUM of field **color** to a list of colors.

```
char * colors[13] =
{
        "Black",        "Charcoal",     "Light Gray",
        "Brown",        "Camel",        "Navy",
        "Light Blue",   "Hunter Green", "Gold",
        "Burgundy",     "Rust",         "White",
        (char *) 0
};
FIELD * color;

set_field_type (color, TYPE_ENUM, colors, FALSE, TRUE);
```

Figure 10-32: Setting a Field to TYPE_ENUM of Colors

Setting the field to TRUE requires the user to enter the seventh character of the color name in certain cases ("Light Blue" and "Light Gray") before a unique match is made.

TYPE_INTEGER

This type enables you to restrict the data in a field to integers.

SYNOPSIS

```
set_field_type (field, TYPE_INTEGER, precision, vmin, vmax);
int precision; /* width for left padding with 0's */
long vmin;     /* minimum acceptable value   */
long vmax;     /* maximum acceptable value   */
```

TYPE_INTEGER takes three additional arguments: a precision specification, a minimum acceptable value, and a maximum acceptable value.

As your end-user enters characters, they are checked for validity. A TYPE_INTEGER value is valid if it consists of an optional minus sign followed by some number of digits. As the end-user tries to leave the field, the range check is applied.
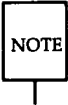
> NOTE
>
> If, contrary to possibility, the maximum value **vmax** is less than or equal to the minimum value **vmin**, the range check is ignored—any integer that fits in the field is valid.

If the range check is passed, the integer is padded on the left with zeros to the precision specification. For instance, if the current value were 18, a precision of 3 would display

018

whereas a precision of 4 would display

0018

For more on ETI's handling of precision, see the manual page **printf**(3S) in the *Programmer's Reference Manual.*

As an example of how to use **set_field_type()** with TYPE_INTEGER, the following might represent a month, padded to 2 digits:

```
FIELD * month;

set_field_type (month, TYPE_INTEGER, 2, 1L, 12L);
/* displays single digit months with leading 0 */
```

Note the requirement that the minimum and maximum values be converted to type **long** with the **L**.

TYPE_NUMERIC

This type restricts the data for the set field to decimal numbers.

```
SYNOPSIS

set_field_type (field, TYPE_NUMERIC, precision, vmin, vmax);
int precision; /* digits to right of the decimal point */
double vmin;   /* minimum acceptable value      */
double vmax;   /* maximum acceptable value      */
```

TYPE_NUMERIC takes three additional arguments: a precision specification, a minimum acceptable value, and a maximum acceptable value.

As your end-user enters characters, they are checked for validity as decimal numbers. A TYPE_NUMERIC value is valid if it consists of an optional minus sign, some number of digits, a decimal point, and some additional digits.

The precision is not used in validation; it is used only in determining the output format. See **printf**(3S) in the *Programmer's Reference Manual* for more on precision. As the end-user tries to leave the field, the range check is applied.

As with TYPE_INTEGER, if the maximum value is less than or equal to the minimum value, the range check is ignored.

For instance, To set a maximum value of $100.00 for a monetary field **amount**, you write:

```
FIELD * amount;

set_field_type (amount, TYPE_NUMERIC, 2, 0.00, 100.00);
```

TYPE_REGEXP

This type enables you to determine whether the data entered into a field matches a specific regular expression.

```
SYNOPSIS

set_field_type (field, TYPE_REGEXP, expression);
char * expression; /* regular expression */
```

TYPE_REGEXP takes one additional argument, the regular expression. See

regcmp(3X) in the *Programmer's Reference Manual* or Chapter 5, **"lex,"** in this Guide for regular expression details.

Consider, for example, how you might create a field that represents a part number with an upper- or lower-case letter followed by exactly 4 digits:

```
FIELD * partnumber;

set_field_type (partnumber, TYPE_REGEXP, "^[A-Za-z][0-9]{4}$");
```

Note that this example assumes the field is 5 characters wide. If not, you may want to change the pattern to accept blanks on either side, thus:

```
FIELD * partnumber;

set_field_type (partnumber, TYPE_REGEXP, "^ *[A-Za-z][0-9]{4} *$");
```

Justifying Data in a Field

Unlike menu items, which always occupy one line, form fields may occupy one or more lines (rows). Fields that occupy one line may be justified left, right, center, or not at all.

```
SYNOPSIS

int set_field_just (field, justification)
FIELD * field;
int justification;

int field_just (field)
FIELD * field;
```

Fields that occupy more than one line are not justified because the data entered typically extends into subsequent lines.

Setting the number of field columns (**cols**) and the minimum width or precision does not always determine where the data fits in the field—there may be excess character space before or after the data. Function **set_field_just()** lets you justify data in one of the following ways:

NO_JUSTIFICATION	- no justification processing (default)
JUSTIFY_LEFT	- left justify value in field
JUSTIFY_RIGHT	- right justify value in field
JUSTIFY_CENTER	- center value in the field

No matter what the justification, fields are automatically left justified as your end-user enters data and edits the field. Once field validation occurs upon the user's request to leave the field, ETI justifies the field as specified.

NOTE   By default, fields are not justified.

For instance, to left justify a name field and right justify an amount field, you can write:

```
FIELD * name, * amount;

set_field_just (name, JUSTIFY_LEFT);  /* left justify a field */

set_field_just (amount, JUSTIFY_RIGHT); /* right justify a field */
```

If successful, **set_field_just()** returns E_OK. If not, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- bad justification

As with most other ETI functions, if one of these functions is passed a NULL field pointer, it assigns or fetches the system default. For instance, to change the system default from no justification to centering the value in its field, you write

```
set_field_just( (FIELD *) 0, JUSTIFY_CENTER); /* set new default */
```

Setting the Field Foreground, Background, and Pad Character

The following functions enable you to set and read the pad character and the low-level ETI (**curses**) attributes associated with your field's foreground and background. The foreground attribute applies only to those field characters that represent data proper, while the background attribute applies to the entire field.

```
SYNOPSIS

int set_field_fore (field, attr)
FIELD * field;
chtype attr;

chtype field_fore (field)
FIELD * field;

int set_field_back (field, attr)
FIELD * field;
chtype attr;

chtype field_back (field)
FIELD * field;

int set_field_pad (field, pad)
FIELD * field;
int pad;

int field_pad (field)
FIELD * field;
```

The initial default for both the foreground and background are A_NORMAL. (See the section on attribute descriptions earlier in this guide or **curses**(3X) in the *Programmer's Reference Manual* for more on screen attributes.) The pad character is the character displayed wherever a blank occurs in the field value stored in field buffer 0.

As an example, to change the background of a field **total** to
A_UNDERLINE and A_STANDOUT, you write

```
FIELD * total;

set_field_back (total, A_UNDERLINE | A_STANDOUT);
```

If function **set_field_fore()** or **set_field_back()** encounter an error, they
return one of the following:

```
E_SYSTEM_ERROR        - system error
E_BAD_ARGUMENT - bad curses attribute
```

The function **set_field_pad()** sets the field's pad character. The default
pad character is a blank. During form processing, pad characters in the field
are translated to blanks in the field's value.

> **NOTE** Because ETI does not distinguish between system-generated pad characters
> and those entered as data, be sure to choose your pad character so as not to
> conflict with valid data.

To set the pad character for field **total** to an asterisk (*), you write

```
FIELD * total;

set_field_pad (total, '*');
```

If successful, function **set_field_pad()** returns E_OK. If not, it returns
one of the following:

```
E_SYSTEM_ERROR        - system error
E_BAD_ARGUMENT        - nonprintable pad character
```

As usual, you can change or access the ETI defaults. To change the
default background to A_UNDERLINE, you write

```
set_field_back ((FIELD *) 0, A_UNDERLINE);
```

Some Helpful Features of Fields

ETI provides special features that promote development of a wide range of form applications. These include field buffers, field status flags, and field user pointers.

Setting and Reading Field Buffers

Recall that you set the number of additional buffers associated with a field upon its creation with **new_field()**. Buffer 0 holds the value of the field. The following functions let you store values in the buffers and later read them.

```
SYNOPSIS

int set_field_buffer (field, buffer, value)
FIELD * field;
int buffer;
char * value;

char * field_buffer (field, buffer)
FIELD * field;
int buffer
```

The parameter **buffer** should range from 0 through **nbuf**, where **nbuf** is the number of additional buffers in the **new_field()** call. All buffers besides 0 may be used to suit your application.

As an example, suppose your application kept a field's default value in field buffer 1. It could use the following code to reset the current field to its default value.

```
#define VAL_BUF            0
#define DFL_BUF            1

void reset_current (form)
FORM * form;
{
        /* set f to current field, described in
           section below, "Manipulating the Current Field" */

        FIELD * f = current_field (form);

        /* set field f to default value */

        set_field_buffer (f, VAL_BUF, field_buffer (f, DFL_BUF));
}
```

If successful, **set_field_buffer()** returns E_OK. If not, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null field pointer, null value, or buffer out of range

Function **field_buffer()**, however, returns NULL if its **field** pointer is NULL or **buffer** is out of range.

The function **field_buffer()** always returns the correct value if the field is not current. However, if the field is current, the function is sometimes inaccurate because data is not moved to field buffer 0 immediately upon entry. You may rest assured that **field_buffer()** is accurate on the current field if

■ it is called from the field check validation routine, if any

■ it is called from the form or field initialization or termination routines, if any

■ it is called just after a REQ_VALIDATION request to the form driver

See the sections below, "Creating a Field Type with Validation Functions," "Establishing Field and Form Initialization and Termination Routines," and "Field Validation Requests," for details on these routines.

Setting and Reading the Field Status

Every field has an associated status flag that is set whenever the field's value (field buffer 0) changes. The following functions enable you to set and access this flag.

```
SYNOPSIS

int set_field_status (field, status)
FIELD * field;
int status;

int field_status (field)
FIELD * field;
```

The field status is TRUE if set or FALSE if cleared. By default, the field status is FALSE when the field is created.

These routines promote increased efficiency where processing need occur only if a field has been changed since some previous state. Two examples are undo operations and database updates. Function **update()** in Figure 10-33, for instance, loops through your field pointer array to save the data in each field if it has been changed (if its **field_status()** is TRUE).

```
void update (form)
FORM * form;
void save_field_data (f)
FIELD * f;
{
        char * data = field_buffer (f, 0);  /* fetch data in field */

        /* save data */

}

{
        FIELD ** f = form_fields (form); /* fetch pointer to field pointer
                                            array */

        while (*f)
        {
                if (field_status (*f))        /* field data changed ? */
                {
                        save_field_data (*f);           /* yes, save new data */
                        set_field_status (*f, FALSE); /* set field status
                                            back */
                }
                ++f;
        }
}
```

Figure 10-33: Using the Field Status to Update a Database

If successful, **set_field_status()** returns E_OK. If not, it returns the following:

      E_SYSTEM_ERROR   - system error

The initial ETI default field status is clear. As always, you can change the default by passing **set_field_status()** a NULL field pointer.

Like the function **field_buffer()**, function **field_status()** always returns the correct value if the field is not current. However, if the field is current, the function is sometimes inaccurate because the status flag is not set immediately. You may rest assured that **field_status()** is accurate on the current field if

■ it is called from the field check validation routine, if any

■ it is called from the form or field initialization or termination routines, if any

■ it is called just after a REQ_VALIDATION request to the form driver

See the sections below, "Creating a Field Type with Validation Functions," "Establishing Field and Form Initialization and Termination Routines," and "Field Validation Requests," for details on these routines.

Setting and Fetching the Field User Pointer

As it does with panels and menus, ETI provides functions to manipulate an arbitrary pointer convenient for field data such as title strings, help messages, and the like.

```
SYNOPSIS

int set_field_userptr (field, userptr)
FIELD * field;
char * userptr;

char * field_userptr (field)
FIELD * field;
```

You can connect an application-defined structure to the field using this pointer. By default, the field user pointer is NULL.

Figure 10-34, for example, shows three routines that use these field functions:

set_field_id()	allocates space for a **struct** ID to be associated with a field and calls **set_field_userptr()** to establish the field's pointer to it
free_field_id()	frees the space for the associated ID
find_match()	searches the names associated with all fields on the form to determine whether any of them match an arbitrary name passed to it

```
#define match(a,b)(strcmp (a, b) == 0)

typedef struct
{
        int             type;
        char *          name;
}
        ID; /* to be hooked onto field userptr */

void set_field_id (f, type, name) /* associate type and name with field f */
FIELD * f;
int type;
char * name;
{
        ID * id = (ID *) malloc (sizeof (ID)); /* allocate space,
                                    see malloc(3X) */

        if (id)                 /* if space allocated */
        {
                id -> type = type;      /* assign type and name */
                id -> name = name;
        }
        set_field_userptr (f, (char *) id);   /* point to id */
}

void free_field_id (f) /* free id connected to field */
FIELD * f;
{
        x = (ID *) field_userptr (*f); /* fetch field user pointer */

        if (x)
                free (x);
```

```
                                                          continued

        }

        FIELD * find_field (f, name) /* find field on form with name */
        FORM * form;
        char * name;
        {
                FIELD ** f = form_fields (form); /* fetch pointer to form's
                                                    field array */
                ID * x;

                while (*f)                      / * for each field in the form */
                {
                        x = (ID *) field_userptr (*f);
                                                /* fetch ID associated with field */

                        if (x && x -> name && match (name, x -> name))
                                                /* does its name match ? */
                                break;
                        ++f;
                }
                return *f;  /* return field pointer of match or NULL */
        }
```

Figure 10-34: Using the Field User Pointer to Match Items

Note that if a match is not found, **find_field()** returns a NULL field pointer. See the previous sections on panel and menu user pointers for more examples.

If successful, **set_field_userptr()** returns E_OK. If not, it returns the following:

      E_SYSTEM_ERROR   - system error

To change the system default user pointer from NULL to one of your choice, you need only pass **set_field_userptr()** a NULL field pointer. Passing a NULL field pointer to **field_userptr()** returns the current default user pointer.

Manipulating Field Options

ETI provides several field options for controlling how data is entered and displayed in a field. The following functions let you set or clear these options or read their settings.

```
SYNOPSIS

int set_field_opts (field, opts)
FIELD * field;
OPTIONS opts;

OPTIONS field_opts (field)
FIELD * field;

options:
        O_VISIBLE
        O_ACTIVE
        O_PUBLIC
        O_EDIT
        O_WRAP
        O_BLANK
        O_AUTOSKIP
        O_NULLOK
        O_PASSOK
```

Function **set_field_opts()** turns off all options that do not appear in its second argument. By default, all options are on.

The field options and their effects are as follows:

O_VISIBLE   determines field visibility. If this option is on, the field is displayed. If this option is off, it is erased. This option is useful for supporting pop-up fields, fields visible or not depending on another field's value.

O_ACTIVE   determines if a field is visited during form processing. If inactivated, the field is ignored during form processing. Inactive fields enable you to create field labels and other static form symbols or changeable symbols that are not affected during form processing.

Examples of fields that change value but are not affected during form processing are row and column totals, as in a spreadsheet program. You can change field values using calls to **set_field_buffer()**.

O_PUBLIC   determines how feedback is presented to the user as data is entered. The data in public fields is displayed as entered, while the data in non-public fields is not displayed at all. Further, in non-public fields, the cursor does not actually move across the field, but the forms subsystem internally maintains the cursor position relative to the field data. You can use non-public fields to implement password fields.

O_EDIT   determines if field editing is permitted. By default, this option is on and a field may be edited. If the O_EDIT option is off, the field may be visited but not changed. Editing requests or attempts to enter data will fail. (REQ_PREV_CHOICE and REQ_NEXT_CHOICE requests, however, are honored, if they are defined for the field's type.) This is useful for creating fields for browsing such as scrollable help messages.

O_WRAP   determines if word wrapping occurs at the end of each line of the field. If any character of the word does not fit on the line as it is entered, the entire word is automatically moved to the beginning of the next line, if there is one. If the O_WRAP option is off, the word is split between the two lines.

O_BLANK   determines if the whole field is automatically erased when the end-user types a character in the first character position of the field before any character position has been changed. If the O_BLANK option is off, this does not occur.

O_AUTOSKIP   determines how the field responds when it becomes full. Ordinarily, when a field is full, an automatic request to move to the next field on the form is generated. If, however, the O_AUTOSKIP option is off, your end-user remains at the end of the field.

O_NULLOK   determines how the field responds when your end-user tries to leave a blank field. By default, this option is on—when a field is blank, a request to leave the field is honored without validating the field. If, on the other hand, the O_NULLOK option is off, the validation procedure is applied to the blank field.

O_PASSOK   When this option is on, the field is checked for validity only if your end-user entered data into the field or edited it. If it is off, the validity check occurs whenever your user leaves the field, whether or not the field was changed. This is useful for fields whose validation function may change dynamically.

Remember that options are Boolean values. So to turn off option O_ACTIVE for field **f0** and to turn it on for field **f1**, you use the Boolean operators and write:

```
FIELD * f0, * f1;

set_field_opts (f0, field_opts (f0) & ~O_ACTIVE); /* turn option off */
set_field_opts (f1, field_opts (f1) | O_ACTIVE); /* turn option on */
```

| NOTE | Although you can change field option settings on posted forms, you cannot change option settings for the current field. |

ETI also provides the following two functions which let you turn a field option on or off without using function **field_opts()**.

SYNOPSIS

```
int field_opts_on (field, opts)
FIELD * field;
OPTIONS opts;

int field_opts_off (field, opts)
FIELD * field;
OPTIONS opts;
```

Unlike function **set_field_opts()**, these functions leave unnamed option set-
tings intact.

As an example, the following code turns options O_BLANK and
O_AUTOSKIP off for field **f0** and on for field **f1**:

```
FIELD * f0, * f1;

field_opts_off (f0, O_BLANK | O_AUTOSKIP); /* turn options off */

field_opts_on  (f1, O_BLANK | O_AUTOSKIP); /* turn options on  */
```

If successful, functions **set_field_opts()**, **field_opts_on()**, and
field_opts_off() return E_OK. If not, they return the following:

E_SYSTEM_ERROR	- system error
E_CURRENT	- cannot change current field options

As usual, you can change the ETI default option settings by passing func-
tion **set_field_options()**, **field_opts_on()**, or **field_opts_off()** a NULL field
pointer. Calling **field_opts()** with a NULL field pointer returns the system
default.

Creating and Freeing Forms

Once you have established a set of fields and their attributes, you are ready to create a form to contain them.

```
SYNOPSIS

FORM * new_form (fields)
FIELD ** fields;

int free_form (form)
FORM * form;
```

The function **new_form()** takes as an argument a NULL-terminated, ordered array of FIELD pointers that define the fields on the form. The order of the field pointers determines the order in which the fields are visited during form driver processing discussed below.

As with the comparable ETI menu function **new_menu()**, function **new_form()** does not copy the array of field pointers. Instead, it saves the pointer to the array. Be sure not to change the array of field pointers once it has been passed to **new_form()**, until the form is freed by **free_form()** or the field array replaced by **set_form_fields()** described in the next section.

Fields passed to **new_form()** are connected to the resulting form.
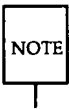
> | NOTE | Fields may be connected to only one form at a time.

To connect fields to another form, you must first disconnect them using **free_form()** or **set_form_fields()**. If **fields** is NULL, the form is created but no fields are connected to it.

Unlike menus, ETI forms are logically divided into pages. Two functions enable you to mark a field that is to start a new page and to return a Boolean value indicating whether a given field does so.

SYNOPSIS

```
int set_new_page(field, bool)
FIELD * field;
int Bool;                /* TRUE or FALSE */

int new_page(field)
FIELD * field;
```

The initial system default value of **new_page()** is FALSE. This means that, unless specified with **set_new_page()**, each field is assumed to continue the current page.

| NOTE | In general, you should make the size of each form page smaller than the form's window size. |

If function **set_new_page()** executes successfully, it returns E_OK. If not, it returns one of the following:

E_SYSTEM_ERROR   -system error
E_CONNECTED   -field connected to form

Figure 10-35 shows how to create a simple two-page form.

```
FIELD * f[7];
FORM * form;

/* create fields as described in section above, "Creating and Freeing Fields" */

f[0] = new_field (...); /* 1st field on page 1 */
f[1] = new_field (...); /* 2nd field on page 1 */
f[2] = new_field (...); /* 3rd field on page 1 */
f[3] = new_field (...); /* 4th field on page 1 */

f[4] = new_field (...); /* 1st field on page 2 */
f[5] = new_field (...); /* 2nd field on page 2 */

f[6] = (FIELD *) 0;  /* signal end of form */

set_new_page (f[4], TRUE); /* start new page with fifth field f[4] */

form = new_form (f); /* create the form */
```

Figure 10-35: Creating a Form

If successful, **new_form()** returns a pointer to the new form. If there is no memory available for the form or one of the given fields is connected to another form, **new_form()** returns NULL. Undefined results occur if the array of field pointers is not NULL-terminated.

The function **free_form()** disconnects all fields and frees any space allocated for the form. Its argument is a form pointer previously obtained from **new_form()**. The fields themselves are not automatically freed.

NOTE | You should free the fields comprising a form using **free_field()** only after you free their form using **free_form()**.

If successful, **free_form()** returns E_OK. If not, it returns one of the following:

      E_SYSTEM_ERROR   - system error
      E_BAD_ARGUMENT   - null form pointer
      E_POSTED   - form is posted

Posting forms is described below.

As with panel, item, menu, and field pointers, form pointers should not be used once they are freed. If they are, undefined results occur.

Manipulating Form Attributes

Recall that an attribute is any feature whose value can be set or read by an appropriate ETI function. A form attribute is any form feature whose value can be set or read by an appropriate ETI function. The set of fields connected to a form and the number of fields connected to it are examples of form attributes.

Changing and Fetching the Fields on an Existing Form

Once you create a form with one set of fields using **new_form()**, you can change the fields connected to it.

```
SYNOPSIS

int set_form_fields (form, fields)
FORM * form;
FIELD ** fields;

FIELD ** form_fields (form)
FORM * form;
```

Like **new_form()**, function **set_form_fields()** takes as an argument a NULL-terminated, ordered array of FIELD pointers that define the fields on the form and determine the order in which the fields are visited during form driver processing.

When **set_form_fields()** is called, the fields previously connected to the form are disconnected from it (but not freed) before the new fields are connected.

Like any set of fields connected to a form, the new fields cannot be passed to other forms while they are connected to the given form. You must first disconnect them by calling **free_form()** or again calling **set_form_fields()**.

There are two ways to disconnect the fields associated with a form without connecting another set of fields to the form:

- ■ you can call **free_form()**
- ■ you can call **set_form_fields()** with **fields** set to NULL

The first method frees the space allocated for the form, whereas the second does not.

To change the fields associated with **form** to those referenced in array pointer **newfields**, you can write

```
FORM * form;
FIELD ** newfields;


set_form_fields (form, newfields); /* associate new set of fields with form */
```

If function **set_form_fields()** encounters an error, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null form pointer
E_POSTED	- form is posted
E_CONNECTED	- connected field

Posting forms is discussed in the section below, "Posting and Unposting Forms".

The function **form_fields()** returns the array of field pointers defining the form's fields. The function returns NULL if no fields are connected to the form or the form pointer is NULL.

Counting the Number of Fields

The following function returns the number of fields connected to the given form.

```
SYNOPSIS

int field_count (form)
FORM * form;
```

If **form** is NULL, **field_count()** returns -1.

As an example, consider the following routine, which determines whether your user is on the last field of the form as numbered in the field pointer array:

```
int on_last_field (form)
FORM * form;
{
        /* fetch number of last field */

        int lastindex = field_count (form) - 1;

        /* determine whether number of current field is the same */

        return field_index (current_field (form)) == lastindex;
}
```

Note the use of functions **field_index()** and **current_field()**, described below in the section, "Manipulating the Current Field."

Changing ETI Form Default Attributes

During form initialization using **new_form()**, all form attributes are assigned default values. As you can with menu attributes, you can change these default attribute values by calling the appropriate function with a NULL form pointer as its first argument. All subsequent forms created using **new_form()** will then have the new default attribute value. However, forms created before the change to the current default value will retain the initial values of their attributes. Several examples of changing default values occur throughout the rest of this chapter.

Displaying Forms

In general, to display a form, you determine the form dimensions, optionally associate a window and subwindow with the form, post the form, and refresh the screen.

Determining the Dimensions of Forms

Every form is associated with a window and subwindow.

| NOTE | By default, (1) the form window is NULL, which by convention means that ETI uses **stdscr** as the form window; and (2) the form subwindow is NULL, which means that ETI uses the form window as the form subwindow. |

Windows are used to create borders, titles, and the like. Before ETI posts a form, it must determine the sizes of its window and subwindow.

To determine the minimum window or subwindow size for a form, ETI considers the following:

■ the number of rows and columns for each field

■ the starting position (upper left corner) of each field within the form subwindow

By automatically fetching this information previously established by calls to **new_field()**, function **scale_form()** saves you the effort of calculating the size of your form subwindow.

Scaling the Form

Considering the above information, this function returns the minimum window size necessary for containing the form.

```
SYNOPSIS

int scale_form (form, rows, cols)
FORM * form;
int * rows;
int * cols;
```

Because function **scale_form()** must return more than one value (namely, the minimum number of rows and columns for the form) and C passes parameters "by value" only, the arguments of **scale_form()** are pointers. The pointer arguments **rows** and **cols** point to locations used to return the minimum number of rows and columns for the form.

| NOTE | You should call **scale_menu()** only after the form's fields have been connected to the form using **new_form()** or **set_form_fields()**. |

As an example, to return the minimum (sub)window size for form **f** in variables **rows** and **cols**, you can write the following:

```
FORM * form;
int rows, cols;

    /* create fields
    create form */

    /* determine minimum row and column size */
scale_form (form, &rows, &cols);

    /* create form subwindow, as described in next section */
```

If function **scale_form()** encounters an error, it returns one of the following:

```
E_SYSTEM_ERROR      - system error
E_BAD_ARGUMENT      - null form pointer
E_NOT_CONNECTED     - no fields connected to the form
```

Associating Windows and Subwindows with a Form

Remember that two windows are associated with every form—the form window and the form subwindow. The following functions assign windows and subwindows to forms and fetch those previously assigned to them.

```
SYNOPSIS

int set_form_win (form, window)
FORM * form;
WINDOW * window;

WINDOW * form_win (form)
FORM * form;

int set_form_sub (form, window)
FORM * form;
WINDOW * window;

WINDOW * form_sub (form)
FORM * form;
```

These functions enable you to place stylistic borders, titles, and other decoration around a form.

> **NOTE** Remember that if the form window is NULL (the default), ETI uses **stdscr**. If the form subwindow is NULL (the default), ETI uses the form window so you need not use functions **set_form_win()** or **set_form_sub()** at all.

If you do not want to use **stdscr**, you should create a window and a subwindow for every form. ETI automatically writes all low-level ETI (**curses**) output of the form proper on the form subwindow. If you want further output (such as borders, titles, or static messages), you should write it on the form window. However, you need not write any further output at all.
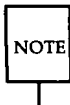
> **NOTE** Be sure to apply all low-level ETI [**curses**(3X)] command output and refresh operations to your form's window, not its subwindow.

Figure 10-36 diagrams the relationship between ETI Form functions, your application program, and its form window and subwindow.
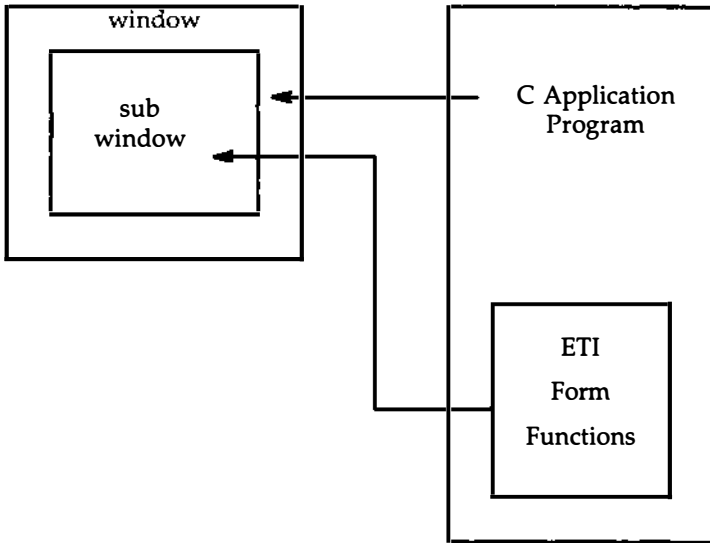
Figure 10-36: Form Functions Write to Subwindow, Application to Window

Figure 10-37 shows how to create a form with a border of the terminal's default vertical and horizontal characters.

```
          /* create window 4 characters larger than form dimensions
          with top left corner at (0, 0). subwindow is positioned
          at (2, 2) relative to the form window origin with dimensions
          equal to the form dimensions. */

FORM * f;
WINDOW * w;
int rows, cols;

scale_form (f, &rows, &cols); /* get dimensions of form */

if (w = newwin (rows+4, cols+4, 0, 0))
{
          set_form_win (f, w);   /* associate window and subwindow with form */

          set_form_sub (f, derwin (w, rows, cols, 2, 2));

          box (w, 0, 0);         /* create border */
}
```

Figure 10-37: Creating a Border Around a Form

───

Function **scale_form()** sets the values of the variables **rows** and **cols**, which provide the form dimensions without the border. Adding four to the dimensions of the form window clearly sets off the form border from the fields of the form (the form proper).

If functions **set_form_win()** or **set_form_sub()** encounter an error, they return one of the following:

E_SYSTEM_ERROR	- system error
E_POSTED	- form is posted

As usual, you can change the default form window or subwindow. For instance, you can change the default form window from **stdscr** to a window **w** by passing a NULL form pointer, as follows:

```
int rows, cols, firstrow, firstcol;

                        /* create form window */

WINDOW * w = newwin (rows, cols, firstrow, firstcol);

set_form_win((FORM *)0, w); /* change default form window to w */
```

Note that if you later change a posted form by writing directly to its window, before continuing you must reposition the form window cursor using **pos_form_cursor()**. See the section below, "Positioning the Form Cursor."

Posting and Unposting Forms

When you have created a form and its window and subwindow, you are ready to post it. To post a form is to display it on the form's subwindow; to unpost a form is to erase it from the form's subwindow.

```
SYNOPSIS

int post_form (form)
FORM * form;

int unpost_form (form)
FORM * form;
```

Unposting a form does not remove its data structure from memory.

NOTE | To post a form, be sure that you have connected fields to it first.

Figure 10-38 uses two application routines, **display_form()** and **erase_form()**, to show how you might post and later unpost a form. The code builds on that used previously in Figure 10-37 to create the form's window and subwindow.

```
static void display_form (f)      /* create form windows and post */
FORM * f;
{
        WINDOW * w;
        int             rows;
        int             cols;

        scale_form (f, &rows, &cols);  /* get dimensions of form */

        /*  create form window as in Figure 10-37  */

        if (w = newwin (rows+4, cols+4, 0, 0))
        {
                set_form_win (f, w);
                set_form_sub (f, derwin (w, rows, cols, 2, 2));
                box (w, 0, 0);
                keypad (w, 1);
        }
        else
                /* error routine in previous section "ETI Low-Level
                 *  Interface (curses) to
                 *  High-Level Functions" */
                error ("error return from newwin", NULL);

        if (post_form (f) != E_OK)      /* post form */

                error ("error return from post_form", NULL);
        else
                refresh (w);
}
static void erase_form (f)    /* unpost and delete form windows */
FORM * f;
{
        WINDOW * w = form_win (f);
        WINDOW * s = form_sub (f);

        unpost_form (f);       /* unpost form */
        werase (w);            /* erase form window */
        wrefresh (w);          /* refresh screen */
        delwin (s);            /* delete form windows */
        delwin (w);
}
```

Figure 10-38: Posting and Unposting a Form

If successful, function **post_form()** returns E_OK. If not, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null form pointer
E_POSTED	- form is already posted
E_NOT_CONNECTED	- no connected fields
E_NO_ROOM	- form does not fit in subwindow

If successful, the function **unpost_form()** returns E_OK. If not, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null form pointer
E_NOT_POSTED	- form is not posted
E_BAD_STATE	- called from init/term function

The initialization and termination routines are discussed in the next section.

Form Driver Processing

Like the function **menu_driver()** for the menu subsystem, function
form_driver() is the workhorse of the form system. Once the form is posted,
the form driver handles all interaction with your end-user. The form driver
responds to

- field navigation requests

- page navigation requests

- field editing requests

- data entry

- field validation requests

Your application passes a character to the form driver for processing and
evaluates the results.

```
SYNOPSIS

int form_driver (form, c)
FORM * form;
int c;
```

As with menu processing, to enable the form driver to process your end-
users' requests, you must write an input key virtualization routine. This rou-
tine defines a correspondence between input keys, control characters, and
escape sequences on the one hand and ETI form requests on the other. The
routine returns a specific form request or application command that the form
driver can process. Upon return from the form driver, your application can
check if the input was processed appropriately. If not, it can specify actions to
be taken. These may include terminating interaction with the form, respond-
ing to help requests, generating an error message, and so on.

Defining the Virtual Key Mapping

For a sample virtual key mapping, consider Figure 10-39, which contains
the application-defined function **get_request()**. Most of the values returned
by **get_request()** are ETI form requests defined in header file **form.h** and
described in the next section. The other values returned (in this example, only
value QUIT) are defined by the application program treated in the later sec-
tion, "Calling the Form Driver."

```
/*  The following key mapping is defined by get_request().
Note that ^X represents the character control-X.

    ^Q          - end form processing

    ^F          - move to next page
    ^B          - move to previous page
    ^N          - move to next field
    ^P          - move to previous field
    home key    - move to first field
    home down   - move to last field
    ^L          - move left to field
    ^R          - move right to field
    ^U          - move up to field
    ^D          - move down to field

    ^W          - move to next word
    ^T          - move to previous word
    ^S          - move to beginning of field data
    ^E          - move to end of field data
    left arrow  - move left in field
    right arrow - move right in field
    down arrow  - move down in field
    up arrow    - move up in field

    ^M <CR>     - enter new line
    ^I          - insert blank character
    ^O          - insert blank line
    ^V          - delete character
    ^H <BS>     - delete previous character
    ^Y          - delete line
    ^G          - delete word
    ^C          - clear to end of line
    ^K          - clear to end of field
    ^X          - clear entire field
    ^A          - request next field choice
    ^Z          - request previous field choice
    ESC         - toggle between insert and overlay mode

    define application commands  */

#define QUIT        (MAX_COMMAND + 1)

static int get_request (w)          /* virtual key mapping */
```

continued

```
WINDOW * w;
{
        static int      mode    = REQ_INS_MODE;
        int             c       = wgetch (w);/* read a character */

        switch (c)
        {
                case 0x11:      /* ^Q */ return  QUIT;

                case 0x06:      /* ^F */ return  REQ_NEXT_PAGE;
                case 0x02:      /* ^B */ return  REQ_PREV_PAGE;
                case 0x0e:      /* ^N */ return  REQ_NEXT_FIELD;
                case 0x10:      /* ^P */ return  REQ_PREV_FIELD;
                case KEY_HOME:                   returnREQ_FIRST_FIELD;
                case KEY_LL:                     returnREQ_LAST_FIELD;
                case 0x0c:      /* ^L */ return  REQ_LEFT_FIELD;
                case 0x12:      /* ^R */ return  REQ_RIGHT_FIELD;
                case 0x15:      /* ^U */ return  REQ_UP_FIELD;
                case 0x04:      /* ^D */ return  REQ_DOWN_FIELD;
                case 0x17:      /* ^W */ return  REQ_NEXT_WORD;
                case 0x14:      /* ^T */ return  REQ_PREV_WORD;
                case 0x13:      /* ^S */ return  REQ_BEG_FIELD;
                case 0x05:      /* ^E */ return  REQ_END_FIELD;
                case KEY_LEFT:                   returnREQ_LEFT_CHAR;
                case KEY_RIGHT:                  returnREQ_RIGHT_CHAR;
                case KEY_DOWN:                   returnREQ_DOWN_CHAR;
                case KEY_UP:                     returnREQ_UP_CHAR;
                case 0x0d:      /* ^M */ return  REQ_NEW_LINE;
                case 0x09:      /* ^I */ return  REQ_INS_CHAR;
                case 0x0f:      /* ^O */ return  REQ_INS_LINE;
                case 0x16:      /* ^V */ return  REQ_DEL_CHAR;
                case 0x08:      /* ^H */ return  REQ_DEL_PREV;
                case 0x19:      /* ^Y */ return  REQ_DEL_LINE;
                case 0x07:      /* ^G */ return  REQ_DEL_WORD;
                case 0x03:      /* ^C */ return  REQ_CLR_EOL;
                case 0x0b:      /* ^K */ return  REQ_CLR_EOF;
                case 0x18:      /* ^X */ return  REQ_CLR_FIELD;
                case 0x01:      /* ^A */ return  REQ_NEXT_CHOICE;
                case 0x1a:      /* ^Z */ return  REQ_PREV_CHOICE;
                case 0x1b:      /* ESC */
                        if (mode == REQ_INS_MODE)
                                return mode = REQ_OVL_MODE;
                        else
```

```
                                                          continued

                                        return mode = REQ_INS_MODE;
           }
           return c;
  }
```

Figure 10-39: A Sample Key Virtualization Routine

In **get_request()**, only a subset of the requests are defined so that the requests your end-user can make are limited. If you like, you can also map two or more keys onto one request. This is helpful where some terminals lack one of the keys in question. In that case, the user can press the other key to the same effect.

Function **get_request()** first sets the data entry mode for the end-user. Here it is set initially to insert mode. The last case statement in the routine enables your end-user to press the escape key ESC to switch to overlay mode. Both modes are discussed in the "Field Editing Requests" section below.

Next, **get_request()** calls **wgetch()** to read a character entered by the user. The **switch()** statement maps the character read onto a specific application command or form request. The application command QUIT appears here as the first case; the other cases map characters onto form requests. Any character that is not an application command or form request is simply returned unchanged—it is treated as data being entered into the current field.

Note that this key mapping assumes your end-user will be using a terminal with arrow keys (KEY_LEFT, KEY_RIGHT, KEY_UP, KEY_DOWN), a home key (KEY_HOME), and a home down key (KEY_LL).

ETI Form Requests

The ETI form subsystem places the following requests at your application program's disposal.

Page Navigation Requests

These requests enable your end-user to navigate or move from page to page on a multi-page form.

REQ_NEXT_PAGE	- move to next page
REQ_PREV_PAGE	- move to previous page
REQ_FIRST_PAGE	- move to first page
REQ_LAST_PAGE	- move to last page

Page navigation requests are cyclic so that

■ the REQ_NEXT_PAGE request from the last page moves to the first page

■ the REQ_PREV_PAGE from the first page moves to the last page

Inter-Field Navigation Requests on the Current Page

These requests enable your end-user to move from field to field on the current page of a single form.

REQ_NEXT_FIELD	- move to next field
REQ_PREV_FIELD	- move to previous field
REQ_FIRST_FIELD	- move to first field
REQ_LAST_FIELD	- move to last field
REQ_SNEXT_FIELD	- move to sorted next field
REQ_SPREV_FIELD	- move to sorted previous field
REQ_SFIRST_FIELD	- move to sorted first field
REQ_SLAST_FIELD	- move to sorted last field
REQ_LEFT_FIELD	- move left to field
REQ_RIGHT_FIELD	- move right to field
REQ_UP_FIELD	- move up to field
REQ_DOWN_FIELD	- move down to field

All field navigation requests are cyclic on the current page so that

■ the REQ_NEXT_FIELD request from the last field on a page moves to the first field on that page

■ the REQ_PREV_FIELD request from the first field on a page moves to the last field on that page

and so forth. The order of the fields in the field array passed to **new_form()** determines the order in which the fields are visited using the REQ_NEXT_FIELD, REQ_PREV_FIELD, REQ_FIRST_FIELD, and REQ_LAST_FIELD requests.

NOTE	Remember that the order of fields in the form array is simply the order in which fields are processed during form processing. This order bears no necessary relation to the order of the fields as they are displayed on the form page.

Your end-user may also move from field to field on the form page in row-major order—left to right, top to bottom. To do so, you use the REQ_SNEXT_FIELD, REQ_SPREV_FIELD, REQ_SFIRST_FIELD, and REQ_SLAST_FIELD requests.

Finally, your end-user can move about in different directions using the REQ_LEFT_FIELD, REQ_RIGHT_FIELD, REQ_UP_FIELD, and REQ_DOWN_FIELD requests. Note that the first character (top left corner) of the field is used to determine where the field is located relative to other fields. This means, for example, that a multi-line field whose first character is on the second row of a form is not on the same row as a field whose first character is on the third row of a form even though the multi-line field may extend below the third row.

Intra-Field Navigation Requests

These requests let your end-user move about inside a field. They may generate implicit scrolling operations on scrollable fields.

REQ_NEXT_CHAR	- move to next character in field
REQ_PREV_CHAR	- move to previous character in field
REQ_NEXT_LINE	- move to next line in field
REQ_PREV_LINE	- move to previous line in field
REQ_NEXT_WORD	- move to next word in field
REQ_PREV_WORD	- move to previous word in field
REQ_BEG_FIELD	- move to beginning of field
REQ_END_FIELD	- move after last character in field
REQ_BEG_LINE	- move to beginning of line
REQ_END_LINE	- move after last character in line
REQ_LEFT_CHAR	- move left in field
REQ_RIGHT_CHAR	- move right in field
REQ_UP_CHAR	- move up in field
REQ_DOWN_CHAR	- move down in field

The effect of these requests is as follows:

■ The REQ_NEXT_CHAR and REQ_PREV_CHAR requests step forward and backward through the field.

■ The REQ_NEXT_LINE and REQ_PREV_LINE requests move the cursor to the beginning of the next and previous line.

■ The REQ_NEXT_WORD and REQ_PREV_WORD requests move the cursor to the beginning of the next or previous word.

■ The REQ_BEG_FIELD places the cursor at the first non-pad character in the field. The REQ_END_FIELD request places the cursor after the last non-pad character in the field. This lets the user easily add characters to the field. If there is no room, it returns the cursor to the start of the field.

■ The REQ_BEG_LINE request places the cursor at the first non-pad character in the current line of the field. The REQ_END_LINE request places the cursor after the last non-pad character in the current line. If there is no room, it returns the cursor to the start of the line.

■ The REQ_LEFT_CHAR, REQ_RIGHT_CHAR, REQ_UP_CHAR, and REQ_DOWN_CHAR requests move one character position in the stated direction.

Field Editing Requests

These requests set the editing mode—insert or overlay.

REQ_INS_MODE	- begin insert mode
REQ_OVL_MODE	- begin overlay mode

In insert mode (the default), all text is inserted at the current cursor position, while all existing text starting at the current cursor position is moved to the right. In overlay mode, text entered by your end-user overlays (replaces) existing text in the field. In both modes, the cursor is advanced one character position as each character is entered.

The following requests provide a complete set of field editing requests.

REQ_NEW_LINE	- new line request
REQ_INS_CHAR	- insert blank character at cursor
REQ_INS_LINE	- insert blank line at cursor
REQ_DEL_CHAR	- delete character at cursor
REQ_DEL_PREV	- delete character before cursor
REQ_DEL_LINE	- delete line at cursor
REQ_DEL_WORD	- delete word at cursor
REQ_CLR_EOL	- clear to end of line
REQ_CLR_EOF	- clear to end of field
REQ_CLR_FIELD	- clear entire field

The effects of REQ_NEW_LINE and REQ_DEL_PREV requests depend on several factors such as the current mode (insert or overlay) and the cursor position within the field.

■ The effects of REQ_NEW_LINE are as follows:

☐ In insert mode—if the cursor is at the beginning of a field or on the last line of a field, the REQ_NEW_LINE request acts like a REQ_NEXT_FIELD request. Otherwise, the REQ_NEW_LINE request inserts a new line after the current line and moves the text on the current line starting at the cursor position to the beginning of the new line. The cursor is moved to the beginning of the new line.

☐ In overlay mode—if the cursor is at the beginning of a field, the REQ_NEW_LINE request acts like a REQ_NEXT_FIELD request. If the cursor is on the last line of a field, the REQ_NEW_LINE request erases all data from the cursor position to the end of the

line and satisfies a REQ_NEXT_FIELD request. Otherwise, the
REQ_NEW_LINE request erases all data from the cursor position
to the end of the line and moves the cursor to the beginning of the
next line.

■ The effects of the REQ_DEL_PREV request is as follows:

□ In insert mode—if the cursor is at the beginning of a field, the
REQ_DEL_PREV request behaves like a REQ_PREV_FIELD
request. If the cursor is at the beginning of a line other than the
first and the text on that line will fit at the end of the preceding
line, the text is moved and the current line is deleted. Otherwise,
the REQ_DEL_PREV request simply deletes the previous character.

□ In overlay mode—if the cursor is positioned at the beginning of a
field, the REQ_DEL_PREV request behaves like a
REQ_PREV_FIELD request. Otherwise, the REQ_DEL_PREV
request simply deletes the previous character.

Because the requests REQ_NEW_LINE and REQ_DEL_PREV automatically
do a request REQ_NEXT_FIELD or REQ_PREV_FIELD as described, they are
said to be overloaded field editing requests. See the remarks on options
O_NL_OVERLOAD and O_BS_OVERLOAD in the section below, "Setting
and Fetching Form Options."

Scrolling Requests

Remember that you specified the number of offscreen rows of a field, if
any, as an argument to **new_field()** when the field was created. The follow-
ing requests enable your program to scroll through fields to display this offs-
creen data.

REQ_SCR_FLINE	- scroll field forward a line
REQ_SCR_BLINE	- scroll field backward a line
REQ_SCR_FPAGE	- scroll field forward a page
REQ_SCR_BPAGE	- scroll field backward a page

In addition, intra-field navigation requests may generate implicit scrolling on
scrollable fields. See the section above, "Intra-Field Navigation Requests."

Field Validation Requests

This request supports field validation for those field types that have it.

REQ_VALIDATION	- validate current field

| NOTE | In general, the ETI form driver automatically performs validation on a field before the user leaves it. (If your user leaves a field, it is valid.) However, before your user terminates interaction with the form, you should make the REQ_VALIDATION request to validate the current field. |

Recall that on current fields, the values returned by functions **field_buffer()** and **field_status()** are sometimes inaccurate. (See the sections above, "Setting and Reading the Field Buffers" and "Setting and Reading the Field Status.") If, however, you make request REQ_VALIDATION immediately before calling these functions, you can be sure that the values they return are accurate—they agree with what your end-user has entered and appears on the screen.

Choice Requests

The following requests enable your user to request the next or previous value of a field type.

REQ_NEXT_CHOICE   - display next field choice
REQ_PREV_CHOICE   - display previous field choice

TYPE_ENUM is the only generic field type that supports these choice requests. In addition, programmer-defined field types may support these requests. See the section above, "Setting Field Type to Ensure Validation," and and the section below, "Creating and Manipulating Programmer-Defined Field Types," for information on these field types.

Application-Defined Commands

Form requests are implemented as integers above the low-level ETI (**curses**) maximum key value KEY_MAX. A symbolic constant MAX_COMMAND is provided so applications can implement their own commands without conflicting with the ETI form or menu subsystems. All ETI system form requests are greater than KEY_MAX and less than or equal to MAX_COMMAND. You should set your application-defined commands to an integer greater than MAX_COMMAND.

Calling the Form Driver

The ETI form driver works very much like the ETI menu driver. As soon as the form driver receives a request, it checks if it is an ETI form request. If so, it performs the request and reports the results. If the request is not an ETI form request, the form driver checks if the character is data, i.e., a printable ascii character. If it is, it enters the character at the current position in the current field. If the character is not recognized as a form request or data, the form driver assumes the character is an application-defined command and returns E_UNKNOWN_COMMAND.

To illustrate a sample design for calling the form driver, we will consider a program that permits interaction with a sweepstakes entry form reproduced in Figure 10-40.


```
+---------------------------------------------+
|                                             |
|              Sweepstakes Entry Form         |
|                                             |
| Last Name           First        Middle     |
| _____           _____      _____   |
|                                             |
| Comments                                    |
| _____   |
| _____   |
| _____   |
| _____   |
|                                             |
+---------------------------------------------+
```

Figure 10-40: Sweepstakes Form Output

You have already seen much of the sweepstakes program in previous examples. Figure 10-41 shows its remaining routines.

```
            /* This program displays a sweepstakes entry form.  */

#include <string.h>
#include <form.h>

static void start_curses()       /* see the section above, "ETI Low_Level
                                    Interface (curses) to High-Level
                                    Functions" */

static void display_form (f) /* create form windows and post */
                                    /* see Figure 10-38 for details */


static void erase_form (f)   /* unpost and delete form windows */
                                    /* see Figure 10-38 for details */

          /* define application commands */

#define QUIT              (MAX_COMMAND + 1)

static int get_request (w)   /* virtual key mapping  see Figure 10-39 */


static int my_driver (form, c)        /* handle application commands */
FORM * form;
int c;
{
        switch (c)
        {
                case QUIT:

        /* validate current field */

                        if (form_driver (form, REQ_VALIDATION) == E_OK)
                                return TRUE;
                        break;
        }
        beep ();   /* signal error */
        return FALSE;
}

main (argc, argv)
int argc;
char * argv[];
{
```

```
WINDOW *      w;
FORM *        form;
FIELD ** f;
FIELD ** make_fields ();
void          free_fields ();
int           c, done = FALSE;

PGM = argv[0];

if (! (form = new_form (make_fields ())))
        error ("error return from new_form", NULL);

start_curses ();
display_form (form);

/* interact with user */

w = form_win (form);

while (! done)
{
        switch (form_driver (form, c = get_request (w)))
        {
                case E_OK:
                        break;
                case E_UNKNOWN_COMMAND:
                        done = my_driver (form, c);
                        break;
                default:
                        beep (); /* signal error */
                        break;
        }
}

/* terminate form processing */

erase_form (form);
end_curses ();
f = form_fields (form);
free_form (form);
free_fields (f);
exit (0);
}
```

continued

```
typedef FIELD *              (* PF_field ) ();

typedef struct                      /* define struct for creation */
{
        PF_field  type;    /* field constructor*/
        int              rows;   /* number of rows*/
        int              cols;   /* number of columns*/
        int              frow;   /* first row*/
        int              fcol;   /* first column*/
        char *           v;      /* field value*/
}
        FIELD_RECORD;

static FIELD * LABEL (x)    /* create a LABEL field */
FIELD_RECORD * x;
{
        FIELD * f = new_field (1, strlen (x->v), x->frow, x->fcol, 0, 0);

        if (f)
        {
                set_field_buffer (f, 0, x->v);
                field_opts_off (f, O_ACTIVE);
        }
        return f;
}

static FIELD * STRING (x)   /* create a STRING field */
FIELD_RECORD * x;
{
        FIELD * f = new_field (x->rows, x->cols, x->frow, x->fcol, 0, 0);

        if (f)
                set_field_back (f, A_UNDERLINE);
        return f;
}

        /* field definitions */

static FIELD_RECORD F [] =
{
        LABEL,        0,     0,      0,11,"Sweepstakes Entry Form",
        LABEL,        0,     0,      2,0,"Last Name",
```

continued

```
            LABEL,         0,     0,      2,20,"First",
            LABEL,         0,     0,      2,34,"Middle",
            LABEL,         0,     0,      5,0,"Comments",
            STRING,        1,     18,     3,0,(char *) 0,
            STRING,        1,     12,     3,20,(char *) 0,
            STRING,        1,     12,     3,34,(char *) 0,
            STRING,        4,     46,     6,0,(char *) 0,
            (PF_field) 0,  0,     0,      0,0,(char *) 0,
};

#define MAX_FIELD      512

static FIELD *                  fields [MAX_FIELD + 1];/* field buffer */

static FIELD ** make_fields ()          /* create the fields */
{
        FIELD ** f = fields;
        int i;

        for (i = 0; i < MAX_FIELD && F[i].type; ++i, ++f)
                *f = (* F[i].type) (& F[i]);

        *f = (FIELD *) 0;
        return fields;
}

static void free_fields (f)     /* free the fields */
FIELD ** f;
{
        while (*f)
                free_field (*f++);
}
```

Figure 10-41: An Example of Form Driver Usage

Function **main()** first calls an application-defined routine **make_fields()** to create the fields and **new_form()** to create the form. Routine **make_fields()**

offers a somewhat different way to create fields from that used in the example in Figure 10-29. (Array **F** holds the string labels and field sizes; it can be changed so that **make_fields()** can create any form.) Function **main()** then initializes **curses** using **start_curses()** and displays the form using **display_form()**.

In its **while** loop, **main()** repeatedly calls **form_driver()** with the character returned by **get_request()**. If the form driver does not recognize the character as a request or data, it returns E_UNKNOWN_COMMAND, whereupon the application-defined routine **my_driver()** is called with the same character. Routine **my_driver()** processes the application-defined commands. In this example, there is only one, QUIT. Note how this request automatically calls the form driver again, now with the REQ_VALIDATION request. Remember that this request is necessary to ensure that current field validation occurs before your end-user leaves the form. If validation is successful, **my_driver()** returns TRUE. In turn, this sets **done** to TRUE, and the while loop is exited.

Finally, **main()** erases the form, terminates low-level ETI (**curses**), frees the form and its fields, and exits the program.

This example is typical, but it is only one of many ways you can structure an application. ETI's flexibility lets you use it over a wide range of applications.

Like other ETI routines that return an **int**, the form driver returns E_OK if it recognizes and processes the input character argument. If it encounters an error, it returns one of the following:

E_SYSTEM_ERROR	-system error
E_BAD_ARGUMENT	-null form pointer
E_BAD_STATE	-called from init/term routines
E_NOT_POSTED	-form is not posted
E_UNKNOWN_COMMAND	-unknown command
E_REQUEST_DENIED	-recognized request failed
E_INVALID_FIELD	-failed field validation

> NOTE Like the menu driver, the form driver may not be called from any of the initialization or termination routines described next. Any attempt to do so returns E_BAD_STATE.

Establishing Field and Form Initialization and Termination Routines

As with the menu driver, you may sometimes want the form driver to execute a specific routine whenever the current field or form changes. The following routines let you do this.

```
SYNOPSIS

typedef void (* PTF_void ) ();

int set_form_init (form, func)
FORM * form;
PTF_void func

PTF_void form_init (form)
FORM * form;

int set_form_term (form, func)
FORM * form;
PTF_void func;

PTF_void form_term (form)
FORM * form;

int set_field_init (form, func)
FORM * form;
PTF_void func

PTF_void field_init (form)
FORM * form;

int set_field_term (form, func)
FORM * form;
PTF_void func;

PTF_void field_term (form)
FORM * form;
```

The argument **func** is a pointer to the specific function you want executed by the form driver. This application-defined function itself takes a form pointer as an argument.

As with menus, if you want your application to execute a routine at one of the initialization or termination points listed below, you should call the appropriate form initialization or termination routine at the start of your program. If you do not want a specific function called in these cases, you may refrain from calling these routines altogether.

Function set_form_init()

The argument **func** to this function is automatically called by the form driver

- when the form is posted

- just after every form page operation, i.e., after the page changes on a posted form

Function set_field_init()

The argument **func** to this function is automatically called by the form driver

- when the form is posted

- just after a field change operation, i.e., every time the current field changes on a posted form

Function set_field_term()

The argument **func** to this function is automatically called by the form driver

- just after the field is validated, i.e., just before the current field changes on a posted form

- when the form is unposted

Function set_form_term()

The argument **func** to this function is automatically called by the form driver

■ just before every form page operation, i.e., just before the page changes on a posted form

■ when the form is unposted

To see more precisely when the initialization and termination routines may be executed, note that your form page and current field can be changed in the following circumstances:

■ Both the form page and the current field may be changed automatically by the form driver in response to a user's request.

■ The form page may be changed when the current field is changed using **set_current_field()**.

■ The current field is changed when the page is changed using **set_form_page()**.

> | NOTE | All of these initialization and termination functions are NULL by default. This means that no function need be called.

These functions promote common operations, such as row or column total updates, display of previously invisible fields, activation of previously inactive fields, and more. As an example, Figure 10-42 shows a field termination routine **update_total()**, which dynamically adjusts a column total field whenever a row field value changes. Function **main()** calls **set_field_term()** to establish **update_total()** as the field termination routine.

```
void update_total (form)
FORM * form;
{
        FIELD ** f = form_fields (form);
        char buf[80];
        double total, atof(); /* atof() converts string to float */

        switch (field_index (current_field (form)))
        {
                case ROW_1:
                case ROW_2:
                case ROW_3:

        /* field buffer() returns field's value as string,
         which atof() converts to float */

                total = atof (field_buffer (f[ROW_1], 0)) + /*calculate total*/
                        atof (field_buffer (f[ROW_2], 0)) +
                        atof (field_buffer (f[ROW_3], 0));

                sprintf (buf, "%.2f", total);
                set_field_buffer (f[TOTAL], 0, buf);
                break;
        }
}

main ()
{
        FORM * form;

        set_field_term (form, update_total); /* establish termination routine */
}
```

Figure 10-42: Sample Termination Routine that Updates a Column Total

Function **set_field_buffer()** sets the column total field to the value **total** stored in **buf**. See the section above, "Setting and Reading Field Buffers", for details on **field_buffer()** and **set_field_buffer()**.

For another example, consider Figure 10-43. It shows a common use for field initialization and termination—highlighting a field when it becomes current and removing the highlight when it is no longer current.

```
void bold_off (form)
FORM * form;
{
        /* remove highlight */

        set_field_back (current_field (form), A_UNDERLINE);
}

void bold_on (form)
FORM * form;
{
        /* highlight field */

   .    set_field_back (current_field (form), A_STANDOUT | A_UNDERLINE);
}

main ()
{
        FORM * form;

        /* establish initialization and termination routines */

        set_field_init (form, bold_on);
        set_field_term (form, bold_off);
}
```

Figure 10-43: Field Initialization and Termination to Highlight Current Field

If functions **set_form_init()**, **set_form_term()**, **set_field_init()**, or **set_field_term()** encounter an error, they return the following:

      E_SYSTEM_ERROR   - system error

As usual, if you want a specific default initialization or termination function for all forms or all fields, you can pass the appropriate set function a NULL form pointer. Passing a NULL form pointer to the access functions returns the current ETI default.

Manipulating the Current Field

The current field is the field where your end-user is positioned on the display screen. It changes as the end-user moves about the form entering or changing data. The cursor rests on the current field. To have your application program set or determine the current field, you use the following functions.

```
SYNOPSIS

int set_current_field (form, field)
FORM * form;
FIELD * field;

FIELD * current_field (form)
FORM * form;

int field_index (field)
FIELD * field;
```

The function **set_current_field()** enables you to set the current field, while function **current_field()** returns the pointer to it. The value returned by **field_index()** is the index to the given field in the field pointer array associated with the connected form. This value is in the range of 0 through **N-1**, where **N** is the total number of fields.

When a form is created by **new_form()** or the fields associated with the form are changed by **set_form_fields()** the current field is automatically set to the first visible, active field on page 0.

```
NOTE
```
Your application program need not call **set_current_field()** unless you want to implement field navigation requests that are not supported by the form driver and discussed in the earlier section, "ETI Form Requests".

Figure 10-44 illustrates the use of these functions. Function
set_first_field() uses **set_current_field()** to set the current field to the first
field in the form's field pointer array. Function **first_field()**, on the other
hand, returns a Boolean value indicating whether the current field is the first
field.

```
int set_first_field (form) /* set current field to first field */
FORM * form;
{
        FIELD ** f = form_fields (form);
        return set_current_field (form, f[0]);
}

int first_field (form) /* check if current field is first field */
FORM * form;
{
        FIELD * f = current_field (form);
        return field_index (f) == 0;
}
```

Figure 10-44: Example Manipulating the Current Field

If function **set_current_field()** encounters an error, it returns one of the
following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null form pointer or field not connected to form
E_BAD_STATE	- called from init/term routines
E_INVALID_FIELD	- current field is invalid on posted form
E_REQUEST_DENIED	- field not active or not visible

The function **current_field()** returns (FIELD *) 0 if given a NULL form
pointer or there are no fields connected to the form.

The function **field_index()** returns -1 if its field pointer argument is NULL or the field is not connected to a form.

Changing the Form Page

Two form functions enable your application program to change to another page on the form or to determine the current page of the form.

```
SYNOPSIS

int set_form_page (form, page)
FORM * form;
int page;

int form_page (form)
FORM * form;
```

Upon execution of **set_form_page()**, the current field is set to the first field on the new page that is visible and active (visited during form driver processing). Variable **page** must be in the range of 0 through **N-1**, where **N** is the total number of pages. The function **form_page()** returns the page number of the page currently visible on the screen.

When function **new_form()** creates a form or function **set_form_fields()** changes the fields associated with a form, the form page is automatically set to 0.

> **NOTE** Your application program need not call **set_form_page()** unless you want to implement page navigation requests that are not supported by the form driver and discussed in the earlier section, "ETI Form Requests."

Figure 10-45 illustrates the use of these functions. Function **set_first_page()** uses **set_form_page()** to change to the first page of the form, while function **first_page()** uses **form_page()** to return a Boolean value indicating whether the first page of the form is currently displayed. Note that the first page is numbered 0.

```
int set_first_page (form) /* set to first form page */
FORM * form;
{
        return set_form_page (form, 0);
}

int first_page (form) /* check if on the first form page */
FORM * form;
{
        return form_page (form) == 0; /* return Boolean */
}
```

Figure 10-45: Example Changing and Checking the Form Page Number

If function **set_form_page()** encounters an error, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null form or page out of range
E_BAD_STATE	- called from init/term routines
E_INVALID_FIELD	- current field is invalid on posted form

The function **form_page()** returns -1 if given a NULL form pointer or there are no fields connected to the form.

Positioning the Form Cursor

As with menu processing, some processing of user form requests may move the cursor from the location required for continued processing by the form driver. This function moves the cursor back to where it belongs.

```
int pos_form_cursor (form)
FORM * form;
```

You need call this function only if your application program changes the

cursor position of the form window.

Figure 10-46 illustrates one use of this function. Function **printpage()** repositions the cursor after it prints the page number in the form window.

```
void printpage (form)
FORM * form;
{
        int           p = form_page (form) + 1;
        WINDOW * w = form_win (form);
        int           rows, cols;
        char          buf[80];

        box (w, 0, 0);                    /* put border around form window */
        getmaxyx (w, rows, cols);    /* fetch window size */
        sprintf (buf, " %d ", p);   /* store next page number */


        wmove (w, (rows-1), ((cols-1)-strlen(buf))/2); /* position cursor */
        waddstr (w, buf);          /* print page number */

        /* position the form cursor for continued form processing */

        pos_form_cursor (form);
}

main ()
{
        FORM * form;

        set_form_init (form, printpage);
}
```

Figure 10-46: Repositioning the Cursor After Printing Page Number

If **pos_form_cursor()** encounters an error, it returns one of the following:

E_SYSTEM_ERROR   - system error
E_BAD_ARGUMENT   - null form pointer
E_NOT_POSTED   - form is not posted

Setting and Fetching the Form User Pointer

As it does for items, menus, and fields, ETI supplies a form user pointer for data such as titles, help messages, and the like. These functions enable you to set the pointer and return its referent.

```
SYNOPSIS

int set_form_userptr (form, userptr)
FORM * form;
char * userptr;

char * form_userptr (form)
FORM * form;
```

You can define a structure to be connected to the form using this pointer. By default, the form user pointer is NULL.

Figure 10-47 illustrates the use of these form user pointer functions to determine whether a given name matches a pattern name. Function **main()** uses **set_form_userptr()** to establish the pattern name, while **compare()** uses **form_userptr()** to fetch the pattern and do the comparison.

```
#define match(a,b) (strcmp (a, b) == 0)

int compare (form, name)
FORM * form;
char * name;
{
        char * s = form_userptr (form); /* fetch pattern string */
        return match (name, s);    /* return Boolean indicating match or not */
}

main ()
{
        FORM * form;
        char * form_name;    /* initialize form_name to desired string */

        set_form_userptr (form, form_name);
/                           * set user pointer to point to string */
}
```

Figure 10-47: Pattern Match Example Using form User Pointer

For more user pointer examples, see the previous sections on item, menu, and field user pointers and the sample programs at the end of this guide.

If successful, **set_form_userptr()** returns E_OK. If not, it returns the following:

E_SYSTEM_ERROR   - system error

As usual, you change the default by passing **set_form_userptr()** a NULL form pointer. So, to change the default user pointer to point to the string "***", you write

```
                /* change default user pointer */
set_form_userptr((form *) 0, "***");
```

Setting and Fetching Form Options

ETI provides form options regulating how specific user requests are handled. These functions enable you to set the options and read their settings.

SYNOPSIS

int set_form_opts (form, opts)
FORM * form;
OPTIONS opts;

OPTIONS form_opts (form)
FORM * form;

options:
   O_NL_OVERLOAD
   O_BS_OVERLOAD

Note that function **set_form_opts()** automatically turns off all form options not referenced in its second argument. By default, all options are on.

The effects of the options are as follows:

O_NL_OVERLOAD determines how a REQ_NEW_LINE request is processed. If O_NL_OVERLOAD is on, the request is overloaded. See the section above, "Field Editing Requests", for a description of overloading. If O_NL_OVERLOAD is off, the REQ_NEW_LINE request behavior depends on whether insert mode is on.

           In insert mode, the REQ_NEW_LINE request first inserts a new line after the current line. It then moves the text on the current line starting at the cursor position to the beginning of the new line. The cursor is repositioned to the beginning of the new line.

           In overlay mode, the REQ_NEW_LINE request erases all data from the cursor position to the end of the line. It then repositions the cursor at the beginning of the next line.

O_BS_OVERLOAD   determines how a REQ_DEL_PREV request is pro-
cessed. If O_BS_OVERLOAD is on, the request is
overloaded. See again the section above, "Field
Editing Requests", for information on overloading.
If O_BS_OVERLOAD is off, the REQ_DEL_PREV
request depends on whether insert mode is on.

In insert mode, if the cursor is at the beginning of
any line except the first and the text on the line
will fit at the end of the previous line, the text is
appended to the previous line and the current line
is deleted. If not, the REQ_DEL_PREV request
simply deletes the previous character, if there is
one. If the cursor is at the first character of the
field, the form driver simply returns
E_REQUEST_DENIED.

In overlay mode, the REQ_DEL_PREV request
simply deletes the previous character, if there is
one.

Options are Boolean values, so you use Boolean operators to turn them on
or off. For example, to turn off option O_NL_OVERLOAD of form **f0** and
turn on the same option of form **f1**, you write

```
FORM * f0, * f1;

set_form_opts (f0, form_opts (f0) & ~O_NL_OVERLOAD); /* turn option off */
set_form_opts (f1, form_opts (f1) |  O_NL_OVERLOAD); /* turn option on  */
```

ETI provides two more functions to turn options on and off.

```
int form_opts_on (form, opts)
FORM * form;
OPTIONS opts;

int form_opts_off (form, opts)
FORM * form;
OPTIONS opts;
```

Unlike function **set_form_opts()**, these functions do not affect options unreferenced in their second argument.

Another way to turn off option O_NL_OVERLOAD on form **f0** and turn it on on form **f1** is to write

```
FORM * f0, * f1;

form_opts_off (f0, O_NL_OVERLOAD); /* turn option off */
form_opts_on  (f1, O_NL_OVERLOAD); /* turn option on  */
```

If functions **set_form_opts()**, **form_opts_off()**, or **form_opts_on()** encounter an error, they return the following:

```
E_SYSTEM_ERROR        -system error
```

To change the current system default from, say, O_NL_OVERLOAD to not-O_NL_OVERLOAD without affecting the O_BS_OVERLOAD option, you write

```
form_opts_off( (FORM *) 0, O_NL_OVERLOAD);
```

Creating and Manipulating Programmer-Defined Field Types

In addition to the wealth of field types that ETI automatically provides, ETI lets you create new field types from old ones. For most applications, you may not need them, but when you do, you will have them.

Building a Field Type from Two Other Field Types

One way to define a new field type is to create one from two existing field types. The function **link_fieldtype()** lets you do this.

SYNOPSIS

```
FIELDTYPE * link_fieldtype(type1,type2)
FIELDTYPE * type1;
FIELDTYPE * type2;
```

The constituent types may be system-defined or programmer-defined types. They may require additional arguments for the later call to **set_field_type()** and may be associated with validation functions or choice functions. Validation functions validate the value in the field, while choice functions enable the user to choose the next or previous value of the field type. See the sections below, "Creating a Field Type with Validation Functions" and "Supporting Next and Previous Choice Functions."

If additional arguments are required for the later call to **set_field_type**, those of **type1** should precede those of **type2**. If there are validation or choice functions associated with the constituent types, the new type first executes the function associated with **type1**. If it is successful, it returns TRUE. If not, the new type executes the function associated with **type2**. Whatever it returns is the value returned by the new type.

As an example, the following code creates a new field type that accepts either a color keyword or an integer between 0 and 255, inclusive:

```
FIELD *f1;

extern char ** colors;

ENUM_OR_INT = link_fieldtype (TYPE_ENUM, TYPE_INTEGER);
                    /* Constituent types are System types
                       described in section "Setting the Field Type
                       to Ensure Validation" */

set_field_type (f1, ENUM_OR_INT, colors, FALSE, FALSE, 0, 0L, 255L);
                    /* create field of field type ENUM_OR_INT */
```

Once you have created the new field type, you can create fields of that type. The last statement here creates field **f1**, which accepts only values of type ENUM_OR_INT.

If an error occurs, **link_fieldtype()** returns the following:

```
NULL    -no available memory
```

Creating a Field Type with Validation Functions

Another way to create a new field type is by specifying

■ a function that validates each character as it is entered into the field

■ a function that validates the entire value entered into the field

■ both

Function **new_fieldtype()** returns your new field type given pointers to these validation functions.

SYNOPSIS

```
typedef int (* PTF_int) ();

FIELDTYPE * new_fieldtype (f_check, c_check)
PTF_int f_check;
PTF_int c_check;
```

The form driver automatically calls the named validation functions during form driver processing.

To create a new field type, you must write at least one of the two valida- tion functions. Function **f_check** is a pointer to a function that takes two arguments: a field pointer and an argument pointer. The argument pointer is treated in the next section. **f_check** is called whenever the end-user tries to leave the field. It should check the field value stored in field buffer 0 and return TRUE if the field is valid or FALSE if not. If the validation function fails, your end-user remains on the offending field.

Function **c_check** is also a pointer to a function that takes two arguments: an integer that represents an ASCII character and an argument pointer. Func- tion **c_check** is called as each character is entered by your end-user. It should check the character for validity and return TRUE if it is and FALSE if not.

Function **new_fieldtype()** is useful for creating field types for specialized applications. For example, Figure 10-48 defines a new field type TYPE_HEX as a hex number between **0x0000** and **0xffff**.

```
#include <ctype.h>
#include <form.h>
extern long strtol ();

#define isblank(c) ((c) == ' ')

static int padding = 4;      /* pad on left to 4 digits */
static long vmin = 0x0000L; /* minimum acceptable value */
static long vmax = 0xffffL; /* maximum acceptable value */

static int fcheck_hex (f, arg)
FIELD * f;
```

```
                                                           continued
        char * arg; /* unnecessary here, discussed in the next section */
        {
            char buf[80];
            char * x = field_buffer (f, 0);
            while (*x && isblank (*x)) ++x;

            if (*x)
            {
                char * t = x;
                while (*x && isxdigit (*x)) ++x;
                while (*x && isblank (*x)) ++x;

                if (! *x)
                {
                    long v = strtol (t, (char **) 0, 16);

                    if (v >= vmin && v <= vmax)
                    {
                        sprintf (buf, "%.*lx", padding, v);
                        set_field_buffer (f, 0, buf);
                        return TRUE;
                    }
                }
            }
            return FALSE;
        }
        static int ccheck_hex (c, arg)
        int c;
        char * arg; /*unnecessary in this example, discussed in the next section*/
        {
            return isxdigit (c);
        }
        FIELDTYPE * TYPE_HEX = new_fieldtype (fcheck_hex, ccheck_hex);
                        /* create new field type */
```

Figure 10-48: Creating a Programmer-Defined Field Type

Later, you assign fields with the field type TYPE_HEX as you do with any

field type and field:

```
FIELD * field;

set_field_type (field, TYPE_HEX);
```

Function **ccheck_hex()** checks that the input character is a valid hexade-cimal digit, while function **fcheck_hex()** examines the field value for valid characters and checks the range. If successful, **fcheck_hex()** pads the field to four digits and returns TRUE. If not, it returns FALSE.

| NOTE | The argument **arg** to functions **f_check** and **c_check** is not used in this version of the TYPE_HEX example because the new type does not require additional arguments to the **set_field_type()** routine. |

If successful, **new_fieldtype()** returns a pointer to the new field type. If either argument to **new_fieldtype()** is a NULL pointer, the corresponding validation is not performed. If no memory is available or both function pointers are NULL, **new_fieldtype()** returns NULL.

Freeing Programmer-Defined Field Types

This function frees any space allocated for a field type created with **new_fieldtype()** or **link_fieldtype()**. Its argument is a field type pointer previously obtained from one of these functions.

```
SYNOPSIS

int free_fieldtype (fieldtype)
FIELDTYPE * fieldtype;
```

You may want to free the field type TYPE_HEX from the previous example once fields of that type have been processed. To do so, you write

```
                              /* create field type TYPE_HEX */
                              create fields of this type
                              free fields of this type */

    free_fieldtype(TYPE_HEX); /* free programmer-defined type */
```

If successful, function **free_fieldtype()** returns E_OK. If an error occurs, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- null field type
E_CONNECTED	- type is connected to one or more fields

Once a field type is freed, you must not use it again. If you do, the effect is undefined.

Supporting Programmer-Defined Field Types

You may want to support some programmer-defined field types with additional arguments or with previous and next choice functions. This section explains how to do so.

Argument Support for Field Types

Some field types may require additional arguments to the **set_field_type()** routine, which sets the field type of a field. Function **set_fieldtype_arg()** takes as arguments pointers to functions that manage storage for the additional arguments.

```
SYNOPSIS

typedef char * (* PTF_charP) ();
typedef void   (* PTF_void) ();

int set_fieldtype_arg (fieldtype, make_arg, copy_arg, free_arg)
FIELDTYPE * fieldtype;
PTF_charP make_arg;
PTF_charP copy_arg;
PTF_void free_arg;
```

You must write the functions referenced by pointers **make_arg**, **copy_arg**, and **free_arg**. These functions should do the following:

make_arg	allocate a structure for the field specific parameters to set_field_type() and return a pointer to the saved data
copy_arg	duplicate the structure created by **make_arg**
free_arg	free any storage allocated by **make_arg** or **copy_arg**

Function **make_arg** is called automatically when your application program calls **set_field_type()**. It takes one argument, a **va_list ***. (See **varargs**(5) for details.) Function **make_arg** in turn should call **va_arg()** for each additional argument to **set_field_type()** associated with the field type. Note that function **va_start()** is called by **set_field_type()** before **make_arg** gains control, while function **va_end()** is called by **set_field_type()** after **make_arg** returns.

Function **make_arg** must allocate space for the information associated with the additional arguments, save the information, and return the pointer to the information cast to a character pointer. It is this character pointer that is the argument **arg** to the other functions associated with the field type, namely **copy_arg**, **free_arg**, **f_check**, **c_check**, **next_choice**, and **prev_choice**.

Function **copy_arg** takes as its sole argument a pointer to existing argument information. It returns a pointer to a copy of this information. Function **free_arg()** takes as its sole argument a pointer to existing argument information. It should free any space allocated by **make_arg**.

Figure 10-49 illustrates how you can add padding and range arguments to our TYPE_HEX defined above.

```
/* TYPE_HEX
     set_field_type (f, TYPE_HEX, padding, vmin, vmax);

     int padding;        for padding with leading zeros
     long vmin;          minimum acceptable value
     long vmax;          maximum acceptable value */

#include <form.h>
#include <ctype.h>
#include <varargs.h>
extern long strtol ();
```

```
#define isblank(c) ((c) == ' ')

typedef struct {
    int     padding;
    long    vmin, vmax;
} HEX;

static char * make_hex (ap)
va_list * ap;
{
    HEX * n = (HEX *) malloc (sizeof (HEX));

    if (n)
    {
        n -> padding = va_arg (*ap, int);
        n -> vmin = va_arg (*ap, long);
        n -> vmax = va_arg (*ap, long);
    }
    return (char *) n;
}
static char * copy_hex (arg)
char * arg;
{
    HEX * n = (HEX *) malloc (sizeof (HEX));
    if (n) *n = *((HEX *) arg);
    return (char *) n;
}
static void free_hex (arg)
char * arg;
{
    free (arg);
}


static int fcheck_hex (f, arg)
FIELD * f;
char * arg;
{
    HEX * n = (HEX *) arg;
    int padding = n -> padding;
    long vmin = n -> vmin;
    long vmax = n -> vmax;
```

continued

```
        char buf[80];
        char * x = field_buffer (f, 0);

        while (*x && isblank (*x)) ++x;

        if (*x)
        {
            char * t = x;

            while (*x && isxdigit (*x)) ++x;
            while (*x && isblank (*x)) ++x;

            if (! *x)
            {
                long v = strtol (t, (char **) 0, 16);

                if (v >= vmin && v <= vmax)
                {
                    sprintf (buf, "%.*lx", padding, v);
                    set_field_buffer (f, 0, buf);
                    return TRUE;
                }
            }
        }
        return FALSE;
}
static int ccheck_hex (c, arg)
int c;
char * arg;
{
    return isxdigit (c);
}
FIELDTYPE * TYPE_HEX = new_fieldtype (fcheck_hex, ccheck_hex);
set_fieldtype_arg (TYPE_HEX, make_hex, copy_hex, free_hex);
```

Figure 10-49: Creating TYPE_HEX with Padding and Range Arguments

Later, to create a field that stores a hex number between 0x0000 and 0xffff, we have

```
set_field_type (field, TYPE_HEX, 4, 0x0000L, 0xffffL);
```

From this example, note that

- Your function **make_arg** (here, **make_hex()**) picks off the additional arguments to **set_field_type()** using **va_arg()**.

- Function **make_hex()** allocates a HEX structure, saves the information provided by the additional arguments, and returns a pointer to the saved information.

- Function **copy_hex()** allocates and copies a HEX structure.

- Function **free_hex()** frees a HEX structure.

- Functions **make_hex()** and **copy_hex()** return NULL if the memory allocation fails.

- Function **check_hex()** uses the argument information to do the necessary padding and range check and returns TRUE if successful.

- ETI's internal caller to **make_hex()** and **copy_hex()** automatically checks that the values (**arg**) returned from the functions are not NULL. Therefore, there is no need for functions (such as **fcheck_hex()**) that use these values to check that they are not NULL.

If successful, function **set_fieldtype_arg()** returns E_OK. If an error occurs, it returns one of the following:

E_SYSTEM_ERROR	- system error
E_BAD_ARGUMENT	- field type, make_arg
	copy_arg, or free_arg is NULL

Supporting Next and Previous Choice Functions

Some field types comprise a set of values from which your user chooses (enters) one. The following functions support those types that have a set of choices.

```
SYNOPSIS

typedef char * (* PTF_charP) ();

int set_fieldtype_choice (type, next_choice, prev_choice)
FIELDTYPE *     type;
PTF_int         next_choice;
PTF_int         prev_choice;

int next_choice(f,arg);
FIELD * f;
char * arg;

int prev_choice(f,arg);
FIELD * f;
char * arg;
```

These functions enable the ETI form driver to support the
REQ_NEXT_CHOICE and REQ_PREV_CHOICE requests mentioned in the
earlier section, "Form Driver Processing".

To support these requests, your application-defined functions **next_choice**
and **prev_choice** must

■ take two arguments: a pointer to the current field and a pointer to the
value **arg** that the **make_arg** function (such as **make_hex()** above)
returned

■ use function **field_buffer()** to read the current value

■ call function **set_field_buffer()** with **buffer** argument 0 to set the next
or previous value

■ return success or failure if there is no logically next or previous value

Both functions can be quite similar.

Figure 10-50 shows an implementation of function **next_choice()** for the
field type TYPE_HEX as defined above, such that REQ_NEXT_CHOICE
increments the current value and REG_PREV_CHOICE decrements the
current value.

```
static int next_hex (f, arg)
FIELD * f;
char * arg;
{
        HEX * n = (HEX *) arg;
        long v = n -> vmin;
        char buf[80];
        char * x = field_buffer (f, 0);

        while (*x && isblank (*x)) ++x;

        if (*x).
        {
                v = strtol (x, (char **) 0, 16);
                if (v >= n -> vmin && v < n -> vmax)
                        ++v;
        }
        sprintf (buf, "%.*lx", n -> padding, v);
        set_field_buffer (f, 0, buf);
        return TRUE;
}
static int prev_hex (f, arg)
FIELD * f;
char * arg;
{
        HEX * n = (HEX *) arg;
        long v = n -> vmax;
        char buf[80];
        char * x = field_buffer (f, 0);

        while (*x && isblank (*x)) ++x;

        if (*x)
        {
                v = strtol (x, (char **) 0, 16);
                if (v > n -> vmin && v <= n -> vmax)
                        --v;
        }
        sprintf (buf, "%.*lx", v -> padding, v);
        set_field_buffer (f, 0, buf);
        return TRUE;
}
                /* associate previous and next choice functions */
set_fieldtype_choice (TYPE_HEX, next_hex, prev_hex);
```

Figure 10-50: Creating a Next Choice Function for a Field Type

If given a blank field, your functions **next_choice** and **prev_choice** should, of course, do something reasonable, such as setting the field to the first or last value of the type.

If function **set_fieldtype_choice()** encounters an error, it returns one of the following:

> E_SYSTEM_ERROR   - system error
> E_BAD_ARGUMENT   - either field type, next_choice
> or prev_choice is null

.

Other ETI Routines

Knowing how to use the basic ETI routines to get output and input and to work with windows, panels, menus, and forms, you can design screen management programs that meet the needs of many users. The ETI library, however, has routines that let you do still more in your program. The following few pages briefly describe some of these routines and what they can help you do—namely, draw simple graphics, use a terminal's soft labels, and work with more than one terminal in a single ETI program.

You should be comfortable using the routines previously discussed and the other routines for I/O and window manipulation discussed on the **curses**(3X) manual page before you try to use the following ETI features.
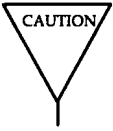
| CAUTION | The routines described under "Routines for Drawing Lines and Other Graphics" and "Routines for Using Soft Labels" are features that are new for UNIX System V Release 3.0 and later releases. If a program uses any of these routines, it may not run on earlier releases of the UNIX System. You must use the Release 3.0 version of the low-level ETI library on UNIX System V Release 3.0 to work with these routines. |

Routines for Drawing Lines and Other Graphics

Many terminals have an alternate character set for drawing simple graphics (or glyphs or graphic symbols). You can use this character set in ETI programs. ETI uses the same names for glyphs as the VT100 line-drawing character set.

To use the alternate character set in an ETI program, you pass a set of variables whose names begin with ACS_ to the ETI routine **waddch()** or a related routine. For example, ACS_ULCORNER is the variable for the upper left corner glyph. If a terminal has a line-drawing character for this glyph, ACS_ULCORNER's value is the terminal's character for that glyph OR'd (|) with the bit-mask A_ALTCHARSET. If no line-drawing character is available for that glyph, a standard ASCII character that approximates the glyph is stored in its place. For example, the default character for ACS_HLINE, a horizontal line, is a – (minus sign). When a close approximation is not available, a + (plus sign) is used. All the standard ACS_ names and their defaults are listed on the **curses**(3X) manual page.

Part of an example program that uses line-drawing characters follows. The example uses the ETI routine **box()** to draw a box around a menu on a screen. **box()** uses the line-drawing characters by default or when | (the pipe) and – are chosen. (See **curses**(3X).) Up and down more indicators are drawn on the box border (using **ACS_UARROW** and **ACS_DARROW**) if the menu contained within the box continues above or below the screen:

```
box(menuwin, ACS_VLINE, ACS_HLINE);
 ...

/* output the up/down arrows */
wmove(menuwin, maxy, maxx - 5);

/* output up arrow or horizontal line */
if (moreabove)
   waddch(menuwin, ACS_UARROW);
else
   addch(menuwin, ACS_HLINE);

/*output down arrow or horizontal line */
if (morebelow)
   waddch(menuwin, ACS_DARROW);
else
   waddch(menuwin, ACS_HLINE);
```

Here's another example. Because a default down arrow (like the lower-case letter v) isn't very discernible on a screen with many lowercase characters on it, you can change it to an uppercase V.

```
if ( ! (ACS_DARROW & A_ALTCHARSET))
   ACS_DARROW = 'V';
```

For more information, see **curses**(3X) in the *Programmer's Reference Manual*.

Routines for Using Soft Labels

Another feature available on most terminals is a set of soft labels across the bottom of their screens. A terminal's soft labels are usually matched with a set of hard function keys on the keyboard. There are usually eight of these labels, each of which is usually eight characters wide and one or two lines high.

The ETI library has routines that provide a uniform model of eight soft labels on the screen. If a terminal does not have soft labels, the bottom line of its screen is converted into a soft label area. It is not necessary for the keyboard to have hard function keys to match the soft labels for an ETI program to make use of them.

Let's briefly discuss most of the ETI routines needed to use soft labels: **slk—init()**, **slk—set()**, **slk—refresh()** and **slk—noutrefresh()**, **slk—clear**, **slk—restore**, **slk—attron()**, **slk—attrset()**, and **slk—attroff()**.

When you use soft labels in an ETI program, you have to call the routine **slk—int()** before **initscr()**. This sets an internal flag for **initscr()** to look at. The flag says to use the soft labels. If **initscr()** discovers that there are fewer than eight soft labels on the screen, that they are smaller than eight characters in size, or that there is no way to program them, then it will remove a line from the bottom of **stdscr** to use for the soft labels. The size of **stdscr** and the **LINES** variable will be reduced by 1 to reflect this change. A properly written program, one that is written to use the **LINES** and **COLS** variables, will continue to run as if the line had never existed on the screen.

slk—init() takes a single argument. It determines how the labels are grouped on the screen should a line get removed from **stdscr**. The choices are between a 3-2-3 arrangement as appears on AT&T terminals, or a 4-4 arrangement as appears on Hewlett-Packard terminals. The ETI routines adjust the width and placement of the labels to maintain the pattern. The widest label generated is eight characters.

The routine **slk—set()** takes three arguments, the label number (1-8), the string to go on the label (up to eight characters), and the justification within the label (0 = left justified, 1 = centered, and 2 = right justified).

The routine **slk—noutrefresh()** is comparable to **wnoutrefresh()** in that it copies the label information onto the internal screen image, but it does not cause the screen to be updated. Since a **wrefresh()** commonly follows, **slk—noutrefresh()** is the function that is most commonly used to output the labels.

Just as **wrefresh()** is equivalent to a **wnoutrefresh()** followed by a **doup-date()**, so too the function **slk_refresh()** is equivalent to a **slk_noutrefresh()** followed by a **doupdate()**.

If **initscr()** removes the bottom line of **stdscr** to simulate soft labels, the routines **slk_attron()**, **slk_attrset()**, and slk_attroff() can be used to manipulate the appearance of the simulated soft labels. Note that these routines will have no effect on soft function key labels supplied by the terminal. These routines are similar to **attron()**, **attrset()**, and **attroff()** (see the section "Simple Input and Output" in this chapter).

To prevent the soft labels from getting in the way of a shell escape, **slk_clear()** may be called before doing the **endwin()**. This clears the soft labels off the screen and does a **doupdate()**. The function **slk_restore()** may be used to restore them to the screen. See the **curses**(3X) manual page for more information about the routines for using soft labels.

Working with More than One Terminal

An ETI program can produce output on more than one terminal at the same time. This is useful for single-process programs that access a common database, such as multi-player games.

Writing programs that output to multiple terminals is a difficult business, and the ETI library does not solve all the problems you might encounter. For instance, the programs—not the library routines—must determine the file name of each terminal line, and what kind of terminal is on each of those lines. The standard method, checking **$TERM** in the environment, does not work, because each process can only examine its own environment.

Another problem you might face is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. However, a program trying to take over another terminal cannot just shut off whatever program is currently running on that line. (Usually, security reasons would also make this inappropriate. But for some applications, such as an inter-terminal communication program or a program that takes over unused terminal lines, it would be appropriate.) A typical solution to this problem requires each user logged in on a line to run a program that notifies a master program that the user is interested in joining the master program and tells it the notification program's process ID, the name of the tty line, and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program and all programs exit.

An ETI program handles multiple terminals by always having a current terminal. All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary ETI routines.

References to terminals in an ETI program have the type **SCREEN***. A new terminal is initialized by calling **newterm**(*type, outfd, infd*). **newterm** returns a screen reference to the terminal being set up. *type* is a character string, naming the kind of terminal being used. *outfd* is a **stdio**(3S) file pointer (**FILE***) used for output to the terminal, and *infd* a file pointer for input from the terminal. This call replaces the normal call to **initscr**(), which calls **newterm(getenv("TERM"), stdout, stdin)**.

To change the current terminal, call **set_term**(*sp*) where *sp* is the screen reference to be made current. **set_term**() returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with **newterm**(). Options such as **cbreak**() and **noecho**() must be set separately for each terminal. The functions **endwin**() and **refresh**() must be called separately for each terminal. Figure 10-51 shows a typical scenario to output a message to several terminals.

```
for (i=0; i<nterm; i++)
{
    set_term(terms[i]);
    mvaddstr(0, 0, "Important message");
    refresh();
}
```

Figure 10-51: Sending a Message to Several Terminals

Working with terminfo Routines

Some programs need to use lower level routines (i.e., primitives) than those offered by the **curses** routines. For such programs, the **terminfo** routines are offered. They do not manage your terminal screen, but rather give you access to strings and capabilities which you can use yourself to manipulate the terminal.

There are three circumstances when it is proper to use **terminfo** routines. The first is when you need only some screen management capabilities, for example, making text standout on a screen. The second is when writing a filter. A typical filter does one transformation on an input stream without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use of the **terminfo** routines is worthwhile. The third is when you are writing a special purpose tool that sends a special purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. Otherwise, you are discouraged from using these routines because the higher level **curses** routines make your program more portable to other UNIX Systems and to a wider class of terminals.

> **NOTE** You are discouraged from using **terminfo** routines except for the purposes noted, because **curses** routines take care of all the glitches present in physical terminals. When you use the **terminfo** routines, you must deal with the glitches yourself. Also, these routines may change and be incompatible with previous releases.

What Every terminfo Program Needs

A **terminfo** program typically includes the header files and routines shown in Figure 10-52.

```
#include <curses.h>
#include <term.h>
...
    setupterm( (char*)0, 1, (int*)0 );
    ...
    putp(clear_screen);
    ...
    reset_shell_mode( );
    exit(0);
```

Figure 10-52: Typical Framework of a **terminfo** Program

The header files <**curses.h**> and <**term.h**> are required because they contain the definitions of the strings, numbers, and flags used by the **terminfo** routines. **setupterm( )** takes care of initialization. Passing this routine the values **(char*)0, 1,** and **(int*)0** invokes reasonable defaults. If **setupterm( )** can't figure out what kind of terminal you are on, it prints an error message and exits. **reset_shell_mode( )** performs functions similar to **endwin( )** and should be called before a **terminfo** program exits.

A global variable like **clear_screen** is defined by the call to **setupterm( )**. It can be output using the **terminfo** routines **putp( )** or **tputs( )**, which gives a user more control. This string should not be directly output to the terminal using the C library routine **printf**(3S), because it contains padding information. A program that directly outputs strings will fail on terminals that require padding or that use the **xon/xoff** flow control protocol.

At the **terminfo** level, the higher level routines like **addch( )** and **getch( )** are not available. It is up to you to output whatever is needed. For a list of capabilities and a description of what they do, see **terminfo**(4); see **curses**(3X) for a list of all the **terminfo** routines.

Compiling and Running a terminfo Program

The general command line for compiling, and the guidelines for running, a program with **terminfo** routines are the same as those for compiling any other **curses** program. See the sections "Compiling an ETI Program" and "Running an ETI Program" in this chapter for more information.

An Example terminfo Program

The example program **termhl** shows a simple use of **terminfo** routines. It is a version of the **highlight** program (see "Program Examples") that does not use the higher level **curses** routines. **termhl** can be used as a filter. It includes the strings to enter bold and underline mode and to turn off all attributes.

```
/*
 * A terminfo level version of the highlight program.
 */

#include <curses.h>
#include <term.h>

int ulmode = 0;    /* Currently underlining */

main(argc, argv)
  int argc;
  char **argv;
{
  FILE *fd;
  int c, c2;
  int outch();

  if (argc > 2)
  {
    fprintf(stderr, "Usage: termhl [file]\n");
    exit(1);
  }

  if (argc == 2)
```

continued

```
{
   fd = fopen(argv[1], "r");
   if (fd == NULL)
   {
      perror(argv[1]);
      exit(2);
   }
}
else
{
   fd = stdin;
}
setupterm((char*)0, 1, (int*)0);

for (;;)
{
   c = getc(fd);
   if (c == EOF)
   break;
   if (c == '\')
   {
      c2 = getc(fd);
      switch (c2)
      {
         case 'B':
         tputs(enter_bold_mode, 1, outch);
         continue;
         case 'U':
         tputs(enter_underline_mode, 1, outch);
         ulmode = 1;
         continue;
         case 'N':
         tputs(exit_attribute_mode, 1, outch);
         ulmode = 0;
         continue;
      }
      putch(c);
      putch(c2);
   }
   else
      putch(c);
```

```
        }
        fclose(fd);
        fflush(stdout);
        resetterm( );
        exit(0);
    }

    /*
     * This function is like putchar, but it checks for underlining.
     */
    putch(c)
        int c;
    {
        outch(c);
        if (ulmode && underline_char)
        {
            outch('\b');
            tputs(underline_char, 1, outch);
        }
    }

    /*
     * Outchar is a function version of putchar that can be passed to
     * tputs as a routine to call.
     */
    outch(c)
        int c;
    {
        putchar(c);
    }
```

Let's discuss the use of the function **tputs(***cap, affcnt, outc***)** in this program
to gain some insight into the **terminfo** routines. **tputs()** applies padding
information. Some terminals have the capability to delay output. Their termi-
nal descriptions in the **terminfo** database probably contain strings like **$<20>**,
which means to pad for 20 milliseconds (see the section "Specify Capabili-
ties" in this chapter). **tputs** generates enough pad characters to delay for the
appropriate time.

tput( ) has three parameters. The first parameter is the string capability to be output. The second is the number of lines affected by the capability. (Some capabilities may require padding that depends on the number of lines affected. For example, **insert_line** may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention *affcnt* is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since *affcnt* is multiplied by the amount of time per item, and anything multiplied by 0 is 0.) The third parameter is a routine to be called with each character.

For many simple programs, *affcnt* is always 1 and *outc* always calls **putchar**. For these programs, the routine **putp**(*cap*) is a convenient abbreviation. **termhl** could be simplified by using **putp( )**.

Now to understand why you should use the **curses**-level routines instead of **terminfo**-level routines whenever possible, note the special check for the **underline_char** capability in this sample program. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. **termhl** keeps track of the current mode, and if the current character is supposed to be underlined, outputs **underline_char**, if necessary. Low-level details such as this are precisely why the **curses** level is recommended over the terminfo level. **curses** takes care of terminals with different methods of underlining and other terminal functions. Programs at the **terminfo** level must handle such details themselves.

termhl was written to illustrate a typical use of the **terminfo** routines. It is more complex than it need be in order to illustrate some properties of **terminfo** programs. The routine **vidattr** [see **curses**(3X)] could have been used instead of directly outputting **enter_bold_mode**, **enter_underline_mode**, and **exit_attribute_mode**. In fact, the program would be more robust if it did, since there are several ways to change video attribute modes.

Working with the terminfo Database

The **terminfo** database describes the many terminals with which **curses** programs, as well as some UNIX System tools, like **vi**(1) (see the *User's/System Administrator's Reference Manual*), can be used. Each terminal description is a compiled file containing the names that the terminal is known by and a group of comma-separated fields describing the actions and capabilities of the terminal. This section describes the **terminfo** database, related support tools, and their relationship to the **curses** library.

Writing Terminal Descriptions

Descriptions of many popular terminals are already described in the **terminfo** database. However, it is possible that you'll want to run a **curses** program on a terminal for which there is not currently a description. In that case, you'll have to build the description.

The general procedure for building a terminal description is as follows:

1. Give the known names of the terminal.

2. Learn about, list, and define the known capabilities.

3. Compile the newly-created description entry.

4. Test the entry for correct operation.

5. Go back to step 2, add more capabilities, and repeat, as necessary.

Building a terminal description is sometimes easier when you build small parts of the description and test them as you go along. These tests can expose deficiencies in the ability to describe the terminal. Also, modifying an existing description of a similar terminal can make the building task easier. (Lest we forget the UNIX System motto: Build on the work of others.)

In the next few pages, we follow each step required to build a terminal description for the fictitious terminal named "myterm."

Name the Terminal

The name of a terminal is the first information given in a **terminfo** terminal description. This string of names, assuming there is more than one name, is separated by pipe symbols (I). The first name given should be the most common abbreviation for the terminal. The last name given should be a long

name that fully identifies the terminal. The long name is usually the manufacturer's formal name for the terminal. All names between the first and last entries should be known synonyms for the terminal name. All names but the formal name should be typed in lowercase letters and contain no blanks. Naturally, the formal name is entered as closely as possible to the manufacturer's name.

Here is the name string from the description of the AT&T Teletype 5420 Buffered Display Terminal:

```
5420|att5420|AT&T Teletype 5420,
```

Notice that the first name is the most commonly used abbreviation and the last is the long name. Also notice the comma at the end of the name string.

Here's the name string for our fictitious terminal, myterm:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
```

Terminal names should follow common naming conventions. These conventions start with a root name, like 5425 or myterm, for example. The root name should not contain odd characters, like hyphens, that may not be recognized as a synonym for the terminal name. Possible hardware modes or user preferences should be shown by adding a hyphen and a 'mode indicator' at the end of the name. For example, the 'wide mode' (which is shown by a **-w**) version of our fictitious terminal would be described as **myterm-w**. **term**(5) describes mode indicators in greater detail.

Learn About the Capabilities

After you complete the string of terminal names for your description, you have to learn about the terminal's capabilities so that you can properly describe them. To learn about the capabilities your terminal has, you should do the following:

- See the owner's manual for your terminal. It should have information about the capabilities available and the character strings that make up the sequence transmitted from the keyboard for each capability.

- Test the keys on your terminal to see what they transmit, if this information is not available in the manual. You can test the keys in one of the following ways — type

    stty -echo; cat -vu
    Type in the keys you want to test;
    for example, see what right arrow (→) transmits.

    <CR>
    <CTRL-D>
    stty echo

  or

    cat >dev/null
    Type in the escape sequences you want to test;
    for example, see what \E[H transmits.
    <CTRL-D>

■ The first line in each of these testing methods sets up the terminal to
carry out the tests. The <CTRL-D> helps return the terminal to its
normal settings.

■ See the **terminfo**(4) manual page. It lists all the capability names you
have to use in a terminal description.

The following section, "Specify Capabilities," gives details.

Specify Capabilities

  Once you know the capabilities of your terminal, you have to describe
them in your terminal description. You describe them with a string of
comma-separated fields that contain the abbreviated **terminfo** name and, in
some cases, the terminal's value for each capability. For example, **bel** is the
abbreviated name for the beeping or ringing capability. On most terminals, a
CTRL-G is the instruction that produces a beeping sound. Therefore, the
beeping capability would be shown in the terminal description as **bel=^G,**.

  The list of capabilities may continue onto multiple lines as long as white
space (that is, tabs and spaces) begins every line but the first of the descrip-
tion. Comments can be included in the description by putting a # at the
beginning of the line.

  The **terminfo**(4) manual page has a complete list of the capabilities you
can use in a terminal description. This list contains the name of the capabil-
ity, the abbreviated name used in the database, the two-letter code that
corresponds to the old **termcap** database name, and a short description of the
capability. The abbreviated name that you will use in your database descrip-
tions is shown in the column titled "Capname."

> NOTE
>
> For a **curses** program to run on any given terminal, its description in the **terminfo** database must include, at least, the capabilities to move a cursor in all four directions and to clear the screen.

A terminal's character sequence (value) for a capability can be a keyed operation (like CTRL-G), a numeric value, or a parameter string containing the sequence of operations required to achieve the particular capability. In a terminal description, certain characters are used after the capability name to show what type of character sequence is required. Explanations of these characters follow:

This shows a numeric value is to follow. This character follows a capability that needs a number as a value. For example, the number of columns is defined as **cols#80,**.

=   This shows that the capability value is the character string that follows. This string instructs the terminal how to act and may actually be a sequence of commands. There are certain characters used in the instruction strings that have special meanings. These special characters follow:

    ˆ   This shows a control character is to be used. For example, the beeping sound is produced by a CTRL-G. This would be shown as ˆG.

\E or \e   These characters followed by another character show an escape instruction. An entry of \EC would transmit to the terminal as ESCAPE-C.

\n   These characters provide a <NL> character sequence.

\l   These characters provide a linefeed character sequence.

\r   These characters provide a return character sequence.

\t   These characters provide a tab character sequence.

\b   These characters provide a backspace character sequence.

\f   These characters provide a formfeed character sequence.

\s   These characters provide a space character sequence.

nnn   These are characters whose three-digit octal is *nnn*, where *nnn* can be one to three digits.

$< >   These symbols are used to show a delay in milliseconds. The desired length of delay is enclosed inside the "less than/greater than" symbols (< >). The amount of delay may be a whole number, a numeric value to one decimal place (tenths), or either form followed by an asterisk (*). The * shows that the delay will be proportional to the number of lines affected by the operation. For example, a 20-millisecond delay per line would appear as $<20*>. See the **terminfo**(4) manual page for more information about delays and padding.

Sometimes, it may be necessary to comment out a capability so that the terminal ignores this particular field. This is done by placing a period (.) in front of the abbreviated name for the capability. For example, if you would like to comment out the beeping capability, the description entry would appear as

.bel=^G,

With this background information about specifying capabilities, let's add the capability string to our description of myterm. We'll consider basic, screen-oriented, keyboard-entered, and parameter string capabilities.

Basic Capabilities

Some capabilities common to most terminals are bells, columns, lines on the screen, and overstriking of characters, if necessary. Suppose our fictitious terminal has these and a few other capabilities, as listed below. Note that the list gives the abbreviated **terminfo** name for each capability in the parentheses following the capability description. The terminal capabilities follow:

■ an automatic wrap around to the beginning of the next line whenever the cursor reaches the right-hand margin (**am**)

■ the ability to produce a beeping sound [the instruction required to produce the beeping sound is ^G (**bel**)].

■ an 80-column wide screen (**cols**)

- a 30-line long screen (**lines**)

- use of xon/xoff protocol (**xon**)

By combining the name string (see the section "Name the Terminal") and the capability descriptions that we now have, we get the following general **terminfo** database entry:

```
myterm|mytm|mine|fancy|terminal|My FANCY terminal,
     am, bel=^G, cols#80, lines#30, xon,
```

Screen-Oriented Capabilities

Screen-oriented capabilities manipulate the contents of a screen. Our example terminal, myterm, has the following screen-oriented capabilities. Again, the abbreviated command associated with the given capability is shown in parentheses.

- A <CR> is a CTRL-M (**cr**).

- A cursor up one line motion is a CTRL-K (**cuu1**).

- A cursor down one line motion is a CTRL-J (**cud1**).

- Moving the cursor to the left one space is a CTRL-H (**cub1**).

- Moving the cursor to the right one space is a CTRL-L (**cuf1**).

- Entering reverse video mode is an ESCAPE-D (**smso**).

- Exiting reverse video mode is an ESCAPE-Z (**rmso**).

- A clear to the end of a line sequence is an ESCAPE-K and should have a 3-millisecond delay (**el**).

- A terminal scrolls when receiving a <NL> at the bottom of a page (**ind**).

The revised terminal description for myterm including these screen-oriented capabilities follows:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
     am, bel=^G, cols#80, lines#30, xon,
     cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
     smso=\ED, rmso=\EZ, el=\EK$<3>, ind=\n,
```

Keyboard-Entered Capabilities

Keyboard-entered capabilities are sequences generated when a key is typed on a terminal keyboard. Most terminals have, at least, a few special keys on their keyboard, such as arrow keys and the backspace key. Our example terminal has several of these keys whose sequences are as follows:

■ The backspace key generates a CTRL-H (**kbs**).

■ The up arrow key generates an ESCAPE-[A (**kcuu1**).

■ The down arrow key generates an ESCAPE-[B (**kcud1**).

■ The right arrow key generates an ESCAPE-[C (**kcuf1**).

■ The left arrow key generates an ESCAPE-[D (**kcub1**).

■ The home key generates an ESCAPE-[H (**khome**).

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
     am, bel=^G, cols#80, lines#30, xon,
     cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
     smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
     kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
     kcub1=\E[D, khome=\E[H,
```

Parameter String Capabilities

Parameter string capabilities are capabilities that can take parameters—for example, those used to position a cursor on a screen or turn on a combination of video modes. To address a cursor, the **cup** capability is used and is passed two parameters: the row and column to address. String capabilities, such as **cup** and set attributes **(sgr)** capabilities, are passed arguments in a **terminfo** program by the **tparm( )** routine.

The arguments to string capabilities are manipulated with special '%' sequences similar to those found in a **printf**(3S) statement. In addition, many of the features found on a simple stack-based RPN calculator are available. **cup**, as noted above, takes two arguments: the row and column. **sgr**, takes nine arguments, one for each of the nine video attributes. See **terminfo**(4) for the list and order of the attributes and further examples of **sgr**.

Our fancy terminal's cursor position sequence requires a row and column to be output as numbers separated by a semicolon, preceded by ESCAPE-[and followed by H. The coordinate numbers are 1-based rather than 0-based. Thus, to move to row 5, column 18, from (0,0), the sequence 'ESCAPE-[6 ; 19 H' would be output.

Integer arguments are pushed onto the stack with a '%p' sequence followed by the argument number, such as '%p2' to push the second argument. A shorthand sequence to increment the first two arguments is '%i'. To output the top number on the stack as a decimal, a '%d' sequence is used, exactly as in **printf**. Our terminal's **cup** sequence is built up as follows:

cup=	Meaning
\E[	output ESCAPE-[
%i	increment the two arguments
%p1	push the 1st argument (the row) onto the stack
%d	output the row as a decimal
;	output a semi-colon
%p2	push the 2nd argument (the column) onto the stack
%d	output the column as a decimal
H	output the trailing letter

or

    cup=\E[%i%p1%d;%p2%dH,

Adding this new information to our database entry for myterm produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
    am, bel=^G, cols#80, lines#30, xon,
    cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
    smso=\ED, rmso=\EZ, el=\EK$<3>, ind=0
    kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
    kcub1=\E[D, khome=\E[H,
    cup=\E[%i%p1%d;%p2%dH,
```

See **terminfo**(4) for more information about parameter string capabilities.

Compile the Description

The **terminfo** database entries are compiled using the **tic** compiler. This compiler translates **terminfo** database entries from the source format into the compiled format.

The source file for the description is usually in a file suffixed with **.ti**. For example, the description of myterm would be in a source file named **myterm.ti**. The compiled description of myterm would usually be placed in **/usr/lib/terminfo/m/myterm**, since the first letter in the description entry is **m**. Links would also be made to synonyms of **myterm**, for example, to **/f/fancy**. If the environment variable **$TERMINFO** were set to a directory and exported before the entry was compiled, the compiled entry would be placed in the **$TERMINFO** directory. All programs using the entry would then look in the new directory for the description file if **$TERMINFO** were set, before looking in the default **/usr/lib/terminfo**. The general format for the **tic** compiler is as follows:

tic [-v] [-c] *file*

The -v option causes the compiler to trace its actions and output information about its progress. The -c option causes a check for errors; it may be combined with the -v option. *file* shows what file is to be compiled. If you want to compile more than one file at the same time, you have to first

use **cat**(1) (see the *User's/System Administrator's Reference Manual*) to join them together. The following command line shows how to compile the **terminfo** source file for our fictitious terminal:

> **tic -v myterm.ti<CR>**
> (The trace information appears as the compilation
> proceeds.)

Refer to the **tic**(1M) manual page in the *User's/System Administrator's Reference Manual* for more information about the compiler.

Test the Description

Let's consider three ways to test a terminal description. First, you can test it by setting the environment variable **$TERMINFO** to the path name of the directory containing the description. If programs run the same on the new terminal as they did on the older known terminals, then the new description is functional.

Second, you can test for correct insert line padding by commenting out **xon** in the description and then editing [using **vi**(1)—see the *User's/System Administrator's Reference Manual*) a large file (over 100 lines) at 9600 baud (if possible), and deleting about 15 lines from the middle of the screen. Type **u** (undo) several times quickly. If the terminal messes up, then more padding is usually required. A similar test can be used for inserting a character.

Third, you can use the **tput**(1) (see the *User's/System Administrator's Reference Manual*) command. This command outputs a string or an integer according to the type of capability being described. If the capability is a Boolean expression, then **tput** sets the exit code (0 for TRUE, 1 for FALSE) and produces no output. The general format for the **tput** command is as follows:

> **tput** [-T*type*] *capname*

The type of terminal you are requesting information about is identified with the **-T***type* option. Usually, this option is not necessary because the default terminal name is taken from the environment variable **$TERM**. The *capname* field is used to show what capability to output from the **terminfo** database.

The following command line shows how to output the "clear screen" character sequence for the terminal being used:

> **tput clear**
> (The screen is cleared.)

The following command line shows how to output the number of columns for the terminal being used:

tput cols
(The number of columns used by the terminal appears here.)

The **tput**(1) manual page found in the *User's/System Administrator's Reference Manual* contains more information on the usage and possible messages associated with this command.

Comparing or Printing terminfo Descriptions

Sometime you may want to compare two terminal descriptions or quickly look at a description without going to the **terminfo** source directory. The **infocmp**(1M) (see the *User's/System Administrator's Reference Manual*) command was designed to help you with both of these tasks. Compare two descriptions of the same terminal; for example,

mkdir /tmp/old /tmp/new
TERMINFO=/tmp/old tic old5420.ti
TERMINFO=/tmp/new tic new5420.ti
infocmp -A /tmp/old -B /tmp/new -d 5420 5420

compares the old and new 5420 entries.

To print out the **terminfo** source for the 5420, type

infocmp -I 5420

Converting a termcap Description to a terminfo Description

CAUTION
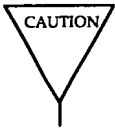
The **terminfo** database is designed to take the place of the **termcap** database. Because of the many programs and processes that have been written with and for the **termcap** database, it is not feasible to do a complete cutover at one time. Any conversion from **termcap** to **terminfo** requires some experience with both databases. All entries into the databases should be handled with extreme caution. These files are important to the operation of your terminal.

The **captoinfo**(1M) (see the *User's/System Administrator's Reference Manual*) command converts **termcap**(4) descriptions to **terminfo**(4) descriptions. When a file is passed to **captoinfo**, it looks for **termcap** descriptions and writes the equivalent **terminfo** descriptions on the standard output. For example,

> captoinfo /etc/termcap

converts the file **/etc/termcap** to **terminfo** source, preserving comments and other extraneous information within the file. The command line

> captoinfo

looks up the current terminal in the **termcap** database, as specified by the $TERM and $TERMCAP environment variables and converts it to **terminfo**.

If you must have both **termcap** and **terminfo** terminal descriptions, keep the **terminfo** description only and use **infocmp –C** to get the **termcap** descriptions.

If you have been using cursor optimization programs with the **–ltermcap** or **–ltermlib** option in the **cc** command line, those programs will still be functional. However, these options should be replaced with the **–lcurses** option.

TAM Transition Library

Character mode applications that run under the Terminal Access Method (TAM) can now run under ETI with a wide range of terminals. This section explains how to use the TAM transition library, the source of this portability. In addition, it explains how you can eventually rewrite your TAM application programs to run more efficiently under ETI without the TAM transition library.

Compiling and Running TAM Applications under ETI

The TAM transition library consists of a header file **tam.h** and a set of library routines. The file **tam.h** translates between TAM routines and equivalent sets of low-level ETI routines. For example, the TAM function **wcreate()** is mapped to the conversion library function **TAMwcreate()**, which consists of a series of low-level ETI calls, such as **newwin()** and **subwin()**.

To use the TAM transition library, be sure to include the standard TAM header file **tam.h** in your application program. So at the beginning of your TAM application program, you should already have

```
#include <tam.h>   /* as usual, for TAM calls */
```

Next, you recompile and link your application program, say **tamprog.c**, to form an executable, as follows:

```
cc -I /usr/add-on/include tamprog.c -ltam -lcurses -o executable_name
```

Note the use of the **-I** option, which tells the compiler where to find the TAM header files. The two uses of the **-l** option link the requisite library subroutines, the TAM transition library and the low-level ETI library.

Alternatively, you might separately compile one or more TAM application files (say, **tam1.c, tam2.c,** and **main.c**) and later link them to form an executable program.

```
cc -c -I /usr/add-on/include tam1.c /* compile files individually */
cc -c -I /usr/add-on/include tam2.c

cc -c -I /usr/add-on/include main.c

                /* link objects to form executable */
cc -o executable_name tam1.o tam2.o main.o -ltam -lcurses
```

Note that the **-I** option is required for the compilation of any file that uses the TAM library.

Tips for Polishing TAM Application Programs Running under ETI

To enable the code in your TAM application program to run smoothly under ETI, you should do the following:

- remove code that would be executed if a low-level **iswind()** function call returned a non-zero value, i.e., *true*. Under the TAM transition library, **iswind()** always returns *false*.

- remove all TAM calls to mouse management routines and the calls **wicon()**, **wicoff()**, and **wrastop()**, because they will translate to null operations.

- remove all machine-specific code, because the TAM transition library does not translate system calls specifically tailored to the UNIX System PC or calls [such as **ioctl**(2)] that have no meaning under ETI. These calls fail under the TAM transition library on all machines except the UNIX System PC.

- note that all calls to **track**(3T) map to the low-level function **wgetc()**.

- remove all references to TAM calls that bear the same name as ETI calls because calls that have the same names in both systems have different effects.

- remove all arbitrary ANSI escape sequences for display output. For example, the TAM transition library does not recognize the escape sequence used on the UNIX System PC in the command **echo** "**\033[J**", which clears the screen. Instead, you should use equivalent ETI routines (here, **clear()**).

Eliminating the superfluous code in the first three cases reduces your program's size and execution time.

How the TAM Transition Library Works

The TAM Transition Library translates between TAM function calls and low-level ETI function calls. It also ensures that escape and control sequences entered at a terminal's keyboard are properly interpreted.

Translations from TAM Calls to ETI Calls

The table in Figure 10-53 summarizes the translation of TAM to low-level ETI (curses) functions. Eventually, if you want to rewrite your TAM applications to make ETI calls directly and to run more efficiently, you can use this table as a guide.

TAM Function	Low-Level ETI (curses(3X)) Equivalent
winit()	Calls initscr().
wexit()	Calls endwin() and exit().
iswind()	Returns FALSE.
wcreate()	Calls newwin() or new_panel().
wdelete()	Calls delwin() or del_panel().
wselect()	Calls touchwin() and wrefresh(), then updates the list of windows to indicate the new ordering.
wgetsel()	Calls top_panel() or bottom_panel() with NULL pointer.
wgetstat()	Calls getyx(), getmaxyx(), or getbegyx().
wsetstat()	Calls del_panel(), then new_panel().
wputc()	Calls waddch().
wputs()	Calls waddstr().
wprintf()	Calls wprintw().
wslk()	Creates small window at bottom and uses **curses** routines with wprintw().
wcmd()	Copies the character string passed by wcmd() to the bottom of the screen.

Figure 10-53: Translations from TAM to ETI Function Calls (Sheet 1 of 4)

TAM Function   Low-Level ETI (curses(3X)) Equivalent

wprompt()	The character string passed by **wprompt()** is copied to the bottom of the screen.
wlabel()	The character string is printed in the upper left corner of the specified window.
wrefresh()	This calls **wrefresh()**. If the window index is -1, all windows should be refreshed in the appropriate order.
wuser()	This functionality is not necessary. Remove this from your code.
wgoto()	This calls **wmove()**.
wgetpos()	This calls **getyx()**.
wgetc()	This calls **wgetch()**. Character translation from ETI to ANSI may be required, depending on the current **keypad** mode.
kcodemap()	This functionality is not necessary. Remove this from your code.
keypad()	This calls **keypad()**.
wsetmouse()	This is a null operation.
wgetmouse()	This is a null operation.
wreadmouse()	This is a null operation.
wprexec()	This calls **erase()** and **refresh()**.
wpostwait()	This calls **wrefresh()** for each window in the window list.
wnl()	The functionality of this routine is not supported by **curses**. See ETI *Release Notes* 1.0 for a workaround.
wicon()	This is a null operation.
wicoff()	This is a null operation.
wrastop()	This is a null operation.
track()	This calls **wgetch()**.
initscr()	This calls **initscr()**.
nl()	The functionality of this routine is not supported by **curses**. See ETI *Release Notes* 1.0 for a workaround.
nonl()	The functionality of this routine is not supported by **curses**. See ETI *Release Notes* 1.0 for a workaround.

Figure 10-53: Translations from TAM to ETI Function Calls (Sheet 2 of 4)

TAM Function	Low-Level ETI (curses(3X)) Equivalent
cbreak()	Calls cbreak().
nocbreak()	Calls nocbreak().
echo()	Calls echo().
noecho()	Calls noecho().
insch()	Calls insch().
getch()	Calls getch().
flushinp()	Calls flushinp().
attron()	Calls attron().
attroff()	Calls attroff().
savetty()	Calls savetty().
resetty()	Calls resetty().
addch()	Calls addch().
addstr()	Calls addstr().
beep()	Calls beep().
clear()	Calls clear().
clearok()	Null operation.
clrtobot()	Calls clrtobot().
clrtoeol()	Calls clrtoeol().
delch()	Calls delch().
deleteln()	Calls deleteln().
erase()	Calls erase().
flash()	Calls flash().
getyx()	Calls wgetyx().
insertln()	Calls insertln().
leaveok()	Null operation.
move()	Calls move().
mvaddch()	Calls move() and addch().
mvaddstr()	Calls move() and addstr().
mvinch()	Calls move() and inch().

Figure 10-53: Translations from TAM to ETI Function Calls (Sheet 3 of 4)

TAM Function   **Low-Level ETI-(curses(3X)) Equivalent**

nodelay()   Calls **nodelay()**.
wndelay()   Calls **nodelay()**.
refresh()   Calls **refresh()**.
resetterm()   Calls **resetterm()**.
baudrate()   Calls **baudrate()**.
endwin()   Calls **endwin()**.
fixterm()   Calls **fixterm()**.
printw()   Calls **printw()**.

Figure 10-53: Translations from TAM to ETI Function Calls (Sheet 4 of 4)

Because the high-level TAM functions in the table in Figure 10-54 make calls only to the low-level functions in the previous table, you can continue to use those high-level TAM functions in your application programs as well. However, with ETI, you cannot use other TAM high-level functions such as **wtargeton()**.

Usable TAM High Level Functions

form()	menu()	message()
pb_empty()	pb_gets()	adf_gttok()
pb_open()	pb_check()	pb_seek()
pb_name()	pb_puts()	pb_weof()
pb_gbuf()	adf_gtwrd()	adf_gtxcd()
wind()	exhelp()	

Figure 10-54: TAM High-Level Functions

The TAM Transition Keyboard Subsystem

Both TAM and ETI use a set of virtual function keys that translate between an escape character sequence entered at the keyboard and a bit pattern inside the machine. Under the TAM transition library, the TAM virtual key values are translated into ETI virtual key values.

The table in Figure 10-55 lists these equivalent virtual key values. Entering the escape sequence listed in the left column will generate the corresponding TAM virtual function key value given in the middle column. The right column lists the ETI equivalent of the TAM virtual key and is for reference only.

TAM Escape Sequence	Virtual Key Value TAM	ETI
ESC-!	s_F1	KEY_F(8)
ESC-@	s_F2	KEY_F(9)
ESC-#	s_F3	KEY_F(10)
ESC-$	s_F4	KEY_F(11)
ESC-%	s_F5	KEY_F(12)
ESC-^	s_F6	KEY_F(13)
ESC-&	s_F7	KEY_F(14)
ESC-*	s_F8	KEY_F(15)
ESC-f1	PF1	KEY_F(16)
ESC-f2	PF2	KEY_F(17)
ESC-f3	PF3	KEY_F(18)

TAM Escape Sequence	TAM	Virtual Key Value ETI
ESC-f4	PF4	KEY_F(19)
ESC-f5	PF5	KEY_F(20)
ESC-f6	PF6	KEY_F(21)
ESC-f7	PF7	KEY_F(22)
ESC-f8	PF8	KEY_F(23)
ESC-f9	PF9	KEY_F(24)
ESC-f0	PF10	KEY_F(25)
ESC-f-	PF11	KEY_F(26)
ESC-f=	PF12	KEY_F(27)
ESC-1	F1	KEY_F(0)
ESC-2	F2	KEY_F(1)
ESC-3	F3	KEY_F(2)
ESC-4	F4	KEY_F(3)
ESC-5	F5	KEY_F(4)
ESC-6	F6	KEY_F(5)
ESC-7	F7	KEY_F(6)
ESC-8	F8	KEY_F(7)
ESC-bg	Beg	KEY_BEG
ESC-BG	s_Beg	KEY_SBEG
ESC-br	Break	KEY_BREAK
ESC-bw	Back	KEY_LEFT
ESC-BW	s_Back	KEY_SLEFT
ESC-ce	Clear	KEY_CLEAR
ESC-CE	Clear	KEY_CLEAR
ESC-ci	ClearLine	KEY_EOL
ESC-CI	s_ClearLine	KEY_SEOL
ESC-cl	Close	KEY_CLOSE
ESC-CL	Close	KEY_CLOSE
ESC-cm	Cmd	KEY_COMMAND
ESC-CM	s_Cmd	KEY_SCOMMAND
ESC-cn	Cancl	KEY_CANCEL
ESC-CN	s_Cancl	KEY_SCANCEL
ESC-cp	Copy	KEY_COPY
ESC-CP	s_Copy	KEY_SCOPY
ESC-cr	Creat	KEY_CREATE
ESC-CR	s_Creat	KEY_SCREATE
ESC-dc	DleteChar	KEY_DC

TAM Escape	Virtual Key Value		
Sequence	TAM		ETI
ESC-Del	DleteChar		KEY_DC
ESC-DC	s_DleteChar		KEY_SDC
ESC-dl	Dlete		KEY_DL
ESC-DL	s_Dlete		KEY_SDL
ESC-dn	Down		KEY_DOWN
ESC-DN	RollDn		KEY_SF
ESC-en	End		KEY_END
ESC-EN	s_End		KEY_SEND
ESC-ESC	Esc		none
ESC-ex	Exit		KEY_EXIT
ESC-EX	s_Exit		KEY_SEXIT
ESC-fi	Find		KEY_FIND
ESC-FI	s_Find		KEY_SFIND
ESC-fw	Forward		KEY_RIGHT
ESC-FW	s_Forward		KEY_SRIGHT
ESC-hl	Help		KEY_HELP
ESC-?	Help		KEY_HELP
ESC-HL	s_Help		KEY_SHELP
ESC-hm	Home		KEY_HOME
ESC-HM	s_Home		KEY_SHOME
ESC-im	InputMode		KEY_IC
ESC-NJ	s_InputMode		KEY_SIC
ESC-mk	Mark		KEY_MARK
ESC-MK	Slect		KEY_SELECT
ESC-ms	Msg		KEY_MESSAGE
ESC-MS	s_Msg		KEY_SMESSAGE
ESC-mv	Move		KEY_MOVE
ESC-MV	s_Move		KEY_SMOVE
ESC-nx	Next		KEY_NEXT
ESC-NX	s_Next		KEY_SNEXT
ESC-op	Open		KEY_OPEN
ESC-OP	Close		KEY_CLOSE
ESC-ot	Opts		KEY_OPTIONS
ESC-OT	s_Opts		KEY_SOPTIONS
ESC-pg	Page		KEY_NPAGE
ESC-PG	s_Page		KEY_PPAGE
ESC-pr	Print		KEY_PRINT

TAM Escape	Virtual Key Value	
Sequence	TAM	ETI
ESC-PR	s_Print	KEY_SPRINT
ESC-pv	Prev	KEY_PREVIOUS
ESC-PV	s_Prev	KEY_SPREVIOUS
ESC-rd	RollDn	KEY_SF
ESC-RD	RollDn	KEY_SF
ESC-re	Ref	KEY_REFERENCE
ESC-RE	Rstrt	KEY_RESTART
ESC-rf	Rfrsh	KEY_REFRESH
ESC-RF	Clear	KEY_CLEAR
ESC-rm	Rsume	KEY_RESUME
ESC-RM	s_Rsume	KEY_SRSUME
ESC-ro	Redo	KEY_REDO
ESC-RO	s_Redo	KEY_SREDO
ESC-rp	Rplac	KEY_REPLACE
ESC-RP	s_Rplac	KEY_SREPLACE
ESC-rs	Rstrt	KEY_REFERENCE
ESC-RS	Rstrt	KEY_RESTART
ESC-ru	RollUp	KEY-SR
ESC-RU	RollUp	KEY_SR
ESC-sl	Slect	KEY_SELECT
ESC-SL	Slect	KEY_SELECT
ESC-ss	Suspd	KEY_SUSPEND
ESC-SS	s_Suspd	KEY_SSUSPEND
ESC-sv	Save	KEY_SAVE
ESC-SV	s_Save	KEY_SSAVE
ESC-ud	Undo	KEY_UNDO
ESC-UD	s_Undo	KEY_SUNDO
ESC-up	Up	KEY_UP
ESC-UP	RollUp	KEY_SR

Figure 10-55: Translation Between TAM Escape Sequences and Virtual Key Values

Some keyboards have one or more keys that emit escape sequences that are identical to the UNIX System PC keyboard sequences but do not match in

terms of functionality. The function of an operationally incompatible key will always map to its **terminfo** specification. The TAM specific function implied by the same escape sequence will be accessible through the technique described above. Mechanisms in **curses**(3X) automatically handle timing conflicts between actual keyboard function keys and UNIX System PC keyboard escape sequences.

Program Examples

The following programs demonstrate uses of low-level ETI (**curses**) functions. See the demonstration programs delivered on the ETI product diskettes for programs that use the high-level ETI functions.

The editor Program

This program illustrates how to use **curses** routines to write a screen editor. For simplicity, **editor** keeps the buffer in **stdscr**; obviously, a real screen editor would have a separate data structure for the buffer. This program has many other simplifications: no provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

Several points about this program are worth making. First, it uses the **move()**, **mvaddstr()**, **flash()**, **wnoutrefresh()** and **clrtoeol()** routines. These routines are all discussed in this chapter under "Working with **curses** Routines."

Second, it also uses some **curses** routines that we have not discussed. For example, the function to write out a file uses the **mvinch()** routine, which returns a character in a window at a given position. The data structure used to write out a file does not keep track of the number of characters in a line or the number of lines in the file, so trailing blanks are eliminated when the file is written. The program also uses the **insch()**, **delch()**, **insertln()**, and **deleteln()** routines. These functions insert and delete a character or line. See **curses**(3X) for more information about these routines.

Third, the editor command interpreter accepts special keys, as well as ASCII characters. On one hand, new users find an editor that handles special keys easier to learn about. For example, it's easier for new users to use the arrow keys to move a cursor than it is to memorize that the letter h means left, j means down, k means up, and l means right. On the other hand, experienced users usually like having the ASCII characters to avoid moving their hands from the home row position to use special keys.

> **NOTE** Because not all terminals have arrow keys, your **curses** programs will work on more terminals if there is an ASCII character associated with each special key.

Fourth, the CTRL-L command illustrates a feature most programs using **curses** routines should have. Often some program beyond the control of the routines writes something to the screen (for instance, a broadcast message) or some line noise affects the screen so much that the routines cannot keep track of it. A user invoking **editor** can type CTRL-L, causing the screen to be cleared and redrawn with a call to **wrefresh(curscr)**.

Finally, another important point is that the input command is terminated by CTRL-D, not the escape key. It is very tempting to use escape as a command, since escape is one of the few special keys available on every keyboard. (Return and break are the only others.) However, using escape as a separate key introduces an ambiguity. Most terminals use sequences of characters beginning with escape (i.e., escape sequences) to control the terminal and have special keys that send escape sequences to the computer. If a computer receives an escape from a terminal, it cannot tell whether the user depressed the escape key or whether a special key was pressed.

editor and other **curses** programs handle the ambiguity by setting a timer. If another character is received during this time, and if that character might be the beginning of a special key, the program reads more input until either a full special key is read, the time out is reached, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press escape, then to type another key quickly, which causes the **curses** program to think a special key has been pressed. Also, a pause occurs until the escape can be passed to the user program, resulting in a slower response to the escape key.

Many existing programs use escape as a fundamental command, which cannot be changed without infuriating a large class of users. These programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a time-out solution. The moral is clear: when designing your **curses** programs, avoid the escape key.

```
/* editor: A screen-oriented editor. The user
 * interface is similar to a subset of vi.
 * The buffer is kept in stdscr to simplify
 * the program.
 */

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

continued

```
#define CTRL(c) ((c) & 037)

main(argc, argv)
int argc;
char **argv;
{
        extern void perror(), exit();
        int i, n, l;
        int c;
        int line = 0;
        FILE *fd;

        if (argc != 2)
        {
                fprintf(stderr, "Usage: %s file\n", argv[0]);
                exit(1);
        }

        fd = fopen(argv[1], "r");
        if (fd == NULL)
        {
                perror(argv[1]);
                exit(2);
        }

        initscr();
        cbreak();
        nonl();
        noecho();
        idlok(stdscr, TRUE);
        keypad(stdscr, TRUE);

        /* Read in the file */
        while ((c = getc(fd)) != EOF)
        {
                if (c == '\n')
                        line++;
                if (line > LINES - 2)
                        break;
                addch(c);
        }
        fclose(fd);
```

```
        move(0,0);
        refresh();
        edit();

        /* Write out the file */
        fd = fopen(argv[1], "w");
        for (l = 0; l < LINES - 1; l++)
        {
                n = len(l);
                for (i = 0; i < n; i++)
                        putc(mvinch(l, i) & A_CHARTEXT, fd);
                putc('\n', fd);
        }
        fclose(fd);
        endwin();
        exit(0);
}

len(lineno)
int lineno;
{
        int linelen = COLS - 1;

        while (linelen >= 0 && mvinch(lineno, linelen) == ' ')
                linelen--;
        return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
        int c;

        for (;;)
```

```
{
        move(row, col);
        refresh();
        c = getch();

        /* Editor commands */
        switch (c)
        {

        /* hjkl and arrow keys: move cursor
         * in direction indicated */
        case 'h':
        case KEY_LEFT:
                if (col > 0)
                        col--;
                else
                        flash();
                break;

        case 'j':
        case KEY_DOWN:
                if (row < LINES - 1)
                        row++;
                else
                        flash();
                break;

        case 'k':
        case KEY_UP:
                if (row > 0)
                        row--;
                else
                        flash();
                break;

        case 'l':
        case KEY_RIGHT:
                if (col < COLS - 1)
                        col++;
                else
                        flash();
                break;
```

continued

```
/* i: enter input mode */
case KEY_IC:
case 'i':
        input();
        break;

/* x: delete current character */
case KEY_DC:
case 'x':
        delch();
        break;

/* o: open up a new line and enter input mode */
case KEY_IL:
case 'o':
        move(++row, col = 0);
        insertln();
        input();
        break;

/* d: delete current line */
case KEY_DL:
case 'd':
        deleteln();
        break;

/* ^L: redraw screen */
case KEY_CLEAR:
case CTRL('L'):
        wrefresh(curscr);
        break;

/* w: write and quit */
case 'w':
        return;
```

continued

```
                /* q: quit without writing */
                case 'q':
                        endwin();
                        exit(2);
                default:
                        flash();
                        break;
                }
        }
}

/*
 * Insert mode: accept characters and insert them.
 * End with ^D or EIC
 */
input()
{
        int c;

        standout();
        mvaddstr(LINES - 1, COLS - 20, "INPUT MODE");
        standend();
        move(row, col);
        refresh();
        for (;;)
        {
                c = getch();
                if (c == CTRL('D') || c == KEY_EIC)
                        break;
                insch(c);
                move(row, ++col);
                refresh();
        }
        move(LINES - 1, COLS - 20);
        clrtoeol();
        move(row, col);
        refresh();
}
```

The highlight **Program**

This program illustrates a use of the routine **attrset()**. **highlight** reads a text file and uses embedded escape sequences to control attributes. \U turns on underlining, \B turns on bold, and \N restores the default output attributes.

Note the first call to **scrollok()**, a routine that we have not previously discussed (see **curses**(3X)). This routine allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw past the bottom of the screen, **scrollok()** automatically scrolls the terminal up a line and calls **refresh()**.

```
/*
 * highlight: a program to turn \U, \B, and
 * \N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */

#include <stdio.h>
#include <curses.h>

main(argc, argv)
int argc;
char **argv;
{
        FILE *fd;
        int c, c2;
        void exit(), perror();

        if (argc != 2)
        {
                fprintf(stderr, "Usage: highlight file\n");
                exit(1);
        }

        fd = fopen(argv[1], "r");

        if (fd == NULL)
```

continued

```
{
        perror(argv[1]);
        exit(2);
}

initscr();
scrollok(stdscr, TRUE);
nonl();
while ((c = getc(fd)) != EOF)
{
        if (c == '\\')
        {
                c2 = getc(fd);
                switch (c2)
                {
                case 'B':
                        attrset(A_BOLD);
                        continue;
                case 'U':
                        attrset(A_UNDERLINE);
                        continue;
                case 'N':
                        attrset(0);
                        continue;
                }
                addch(c);
                addch(c2);
        }
        else
                addch(c);
}
fclose(fd);
refresh();
endwin();
exit(0);
}
```

The scatter Program

This program takes the first **LINES – 1** lines of characters from the standard input and displays the characters on a terminal screen in a random order. For this program to work properly, the input file should not contain tabs or non-printing characters.

```
/*
 *       The scatter program.
 */

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

extern time_t time();

#define MAXLINES 120
#define MAXCOLS  160
char s[MAXLINES][MAXCOLS];          /* Screen Array */
int  T[MAXLINES][MAXCOLS];          /* Tag Array - Keeps track of  *
                                     * the number of characters    *
                                     * printed and their positions. */

main()
{
        register int row = 0,col = 0;
        register int c;
        int char_count = 0;
        time_t t;
        void exit(), srand();

        initscr();
        for(row = 0;row < MAXLINES;row++)
                for(col = 0;col < MAXCOLS;col++)
                        s[row][col]=' ';

        col = row = 0;
        /* Read screen in */
        while ((c=getchar()) != EOF && row < LINES ) {

                if(c != '\n')
```

```
                {
                        /* Place char in screen array */
                        s[row][col++] = c;
                        if(c != ' ')
                                char_count++;
                }
                else
                {
                        col = 0;
                        row++;
                }
        }

        time(&t);/* Seed the random number generator */
        srand((unsigned)t);

        while (char_count)
        {
                row = rand() % LINES;
                col = (rand() >> 2) % COLS;
                if (T[row][col] != 1 && s[row][col] != ' ')
                {
                        move(row, col);
                        addch(s[row][col]);
                        T[row][col] = 1;
                        char_count--;
                        refresh();
                }
        }
        endwin();
        exit(0);
}
```

The show **Program**

show pages through a file, showing one screen of its contents each time you depress the space bar. The program calls **cbreak()** so that you can depress the space bar without having to hit return; it calls **noecho()** to prevent the space from echoing on the screen. The **nonl()** routine, which we have not previously discussed, is called to enable more cursor optimization. The **idlok()** routine, which we also have not discussed, is called to allow insert and delete line. (See **curses**(3X) for more information about these routines). Also notice that **clrtoeol()** and **clrtobot()** are called.

By creating an input file for **show** made up of screen-sized (about 24 lines) pages, each varying slightly from the previous page, nearly any exercise for a **curses()** program can be created. This type of input file is called a show script.

```
#include <curses.h>
#include <signal.h>

main(argc, argv)
int argc;
char *argv[];
{
        FILE *fd;
        char linebuf[BUFSIZ];
        int line;
        void done(), perror(), exit();

        if (argc != 2)
        {
                fprintf(stderr, "usage: %s file\n", argv[0]);
                exit(1);
        }

        if ((fd=fopen(argv[1], "r")) == NULL)
        {

                perror(argv[1]);
                exit(2);
        }
```

```
        signal(SIGINT, done);

        initscr();
        noecho();
        cbreak();
        nonl();
        idlok(stdscr, TRUE);

        while(1)
        {
                move(0,0);
                for (line = 0; line < LINES; line++)
                {
                        if (!fgets(linebuf, sizeof linebuf, fd))
                        {
                                clrtobot();
                                done();
                        }
                        move(line, 0);
                        printw("%s", linebuf);
                }
                refresh();
                if (getch() == 'q')
                        done();
        }
}

void done()
{
        move(LINES - 1, 0);
        clrtoeol();
        refresh();
        endwin();
        exit(0);
}
```

The two **Program**

This program pages through a file, writing one page to the terminal from which the program is invoked and the next page to the terminal named on the command line. It then waits for a space to be typed on either terminal and writes the next page to the terminal at which the space is typed.

two is just a simple example of a two-terminal **curses** program. It does not handle notification; instead, it requires the name and type of the second terminal on the command line. As written, the command **"sleep 100000"** must be typed at the second terminal to put it to sleep while the program runs, and the user of the first terminal must have both read and write permission on the second terminal.

```
#include <curses.h>
#include <signal.h>

SCREEN *me, *you;
SCREEN *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
int argc;
char **argv;
{
        void done(), exit();
        unsigned sleep();
        char *getenv();
        int c;

        if (argc != 4)
        {
            fprintf(stderr, "Usage: two othertty otherttytype inputfile\n");
            exit(1);
        }
```

```
                                                          continued
    fd = fopen(argv[3], "r");
    fdyou = fopen(argv[1], "w+");
    signal(SIGINT, done);       /* die gracefully */

    me = newterm(getenv("TERM"), stdout, stdin);  /* initialize my tty */
    you = newterm(argv[2], fdyou, fdyou);/* Initialize the other terminal */

    set_term(me);       /* Set modes for my terminal */
    noecho();/* turn off tty echo */
    cbreak();/* enter cbreak mode */
    nonl();             /* Allow linefeed */
    nodelay(stdscr, TRUE);      /* No hang on input */

    set_term(you);      /* Set modes for other terminal */
    noecho();
    cbreak();
    nonl();
    nodelay(stdscr,TRUE);

    /* Dump first screen full on my terminal */
    dump_page(me);

    /* Dump second screen full on the other terminal */
    dump_page(you);

    for (;;) /* for each screen full */
    {
        set_term(me);
        c = getch();
        if (c == 'q')/* wait for user to read it */
        done();
        if (c == ' ')
        dump_page(me);

        set_term(you);
        c = getch();
        if (c == 'q')/* wait for user to read it */
        done();
        if (c == ' ')
        dump_page(you);
        sleep(1);
    }
}
```

```
dump_page(term)
  SCREEN *term;
{
        int line;

        set_term(term);
        move(0, 0);
        for (line = 0; line < LINES - 1; line++) {
            if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
            clrtobot();
            done();
            }
            mvaddstr(line, 0, linebuf);
        }
        standout();
        mvprintw(LINES - 1, 0, "--More--");
        standend();
        refresh();        /* sync screen */
}
/*
 * Clean up and exit.
 */
void done()
{
        /* Clean up first terminal */
        set_term(you);
        move(LINES - 1,0);/* to lower left corner */

        clrtoeol();       /* clear bottom line */
        refresh();        /* flush out everything */
        endwin();/* curses cleanup */

        /* Clean up second terminal */
        set_term(me);
        move(LINES - 1,0);/* to lower left corner */
        clrtoeol();       /* clear bottom line */
        refresh();        /* flush out everything */
        endwin();/* curses cleanup */
        exit(0);
}
```

The window **Program**

This example program demonstrates the use of multiple windows. The main display is kept in **stdscr**. When you want to put something other than what is in **stdscr** on the physical terminal screen temporarily, a new window is created covering part of the screen. A call to **wrefresh**() for that window causes it to be written over the **stdscr** image on the terminal screen. Calling **refresh**() on **stdscr** results in the original window being redrawn on the screen. Note the calls to the **touchwin**() routine (which we have not discussed — see **curses**(3X)) that occur before writing out a window over an existing window on the terminal screen. This routine prevents screen optimization in a **curses** program. If you have trouble refreshing a new window that overlaps an old window, it may be necessary to call **touchwin**() for the new window to get it completely written out.

```
#include <curses.h>

WINDOW *cmdwin;

main()

{
        int i, c;
        char buf[120];
        void exit();

        initscr();
        nonl();
        noecho();
        cbreak();

        cmdwin = newwin(3, COLS, 0, 0);/* top 3 lines */
        for (i = 0; i < LINES; i++)
                mvprintw(i, 0, "This is line %d of stdscr", i);
```

continued

```
for (;;)

{
        refresh();
        c = getch();
        switch (c)

        {

        case 'c':/* Enter command from keyboard */
                werase(cmdwin);
                wprintw(cmdwin, "Enter command:");
                wmove(cmdwin, 2, 0);
                for (i = 0; i < COLS; i++)
                        waddch(cmdwin, '-');
                wmove(cmdwin, 1, 0);
                touchwin(cmdwin);
                wrefresh(cmdwin);
                wgetstr(cmdwin, buf);
                touchwin(stdscr);

                /*
                 * The command is now in buf.
                 * It should be processed here.
                 */

        case 'q':
                endwin();
                exit(0);
        }

    }
}
```

The colors **Program**

This program creates two windows. All characters displayed in the first window will be in red, on a blue background. All characters displayed in the second window will be in yellow, on a magenta background.

```
#include <curses.h>

#define  PAIR1     1
#define  PAIR2     2

main( )
{
        WINDOW *win1, *win2;

        intscr( );
        if ((start_color( )) == OK)
        {
                /* create windows */

                win1 = newwin (5, 40, 0, 0);
                win2 = newwin (5, 40, 15, 40);

                /* create two color pairs */

                init_pair (PAIR1, COLOR_RED, COLOR_BLUE);
                init_pair (PAIR2, COLOR_YELLOW, COLOR_MAGENTA);

                /* turn on color attributes for each window */

                wattron (win1, COLOR_PAIR (PAIR1));
                wattron (win2, COLOR_PAIR (PAIR2));

                /* print some text in each window and exit */

                waddstr (win1, "This should be red on blue");
                waddstr (win2, "This should be yellow on magenta");
                wrefresh (win1);
                wrefresh (win2);

                /* wait for any key before terminating */

                wgetch (win2);
        }

        endwin( );
}
```

11 Common Object File Format (COFF)

The Common Object File Format (COFF)

This section describes the Common Object File Format (COFF) used on your computer with the UNIX Operating System. COFF is the format of the output file produced by the assembler, **as**, and the link editor, **ld**.

The following are some key features of COFF:

■ applications can add system-dependent information to the object file without causing access utilities to become obsolete

■ space is provided for symbolic information used by debuggers and other applications

■ programmers can modify the way the object file is constructed by providing directives at compile time

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

■ a file header

■ optional header information

■ a table of section headers

■ data corresponding to the section headers

■ relocation information

■ line numbers

■ a symbol table

■ a string table

Figure 11-1 shows the overall structure.

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

Figure 11-1: Object File Format

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

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

Definitions and Conventions

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

Sections

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

| NOTE | It a mistake to assume that every COFF file will have a certain number of sections, or to assume characteristics of sections such as their order, their location in the object file, or the address at which they are to be loaded. This information is available only after the object file has been created. Programs manipulating COFF files should obtain it from file and section headers in the file. |

Physical and Virtual Addresses

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

Target Machine

Compilers and link editors produce executable object files that are intended to be run on a particular computer. In the case of cross-compilers, the compilation and link editing are done on one computer, with the intent of creating an object file that can be executed on another computer. The term, target machine, refers to the computer on which the object file is destined to run. In the majority of cases, the target machine is the exact same computer on which the object file is being created.

File Header

The file header contains the 20 bytes of information shown in Figure 11-2. The last 2 bytes are flags that are used by **ld** and object file utilities.

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

Figure 11-2: File Header Contents

Magic Numbers

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

Flags

The last 2 bytes of the file header are flags that describe the type of the object file. Currently defined flags are found in the header file **filehdr.h** and are shown in Figure 11-3.

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

Figure 11-3: File Header Flags

File Header Declaration

The C structure declaration for the file header is given in Figure 11-4. This declaration may be found in the header file **filehdr.h**.

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

    long             f_timdat;  /* time and date stamp */

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

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

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

    unsigned short   f_flags;   /* flags */
};

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

Figure 11-4: File Header Declaration

Optional Header Information

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

Standard UNIX System a.out Header

By default, files produced by the link editor for a UNIX System always have a standard UNIX System **a.out** header in the optional header field. The UNIX System **a.out** header is 28 bytes. The fields of the optional header are described in Figure 11-5.

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

Figure 11-5: Optional Header Contents

Whereas the magic number in the file header specifies the machine on which the object file runs, the magic number in the optional header supplies information telling the operating system on that machine how that file should be executed. The magic numbers recognized by the UNIX System V/386 Release 3.2 operating system are given in Figure 11-6.

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

Figure 11-6: UNIX System Magic Numbers

Optional Header Declaration

The C language structure declaration currently used for the UNIX System **a.out** file header is given in Figure 11-7. This declaration may be found in the header file **aouthdr.h**.

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

                                /* to full word boundary */

        long    dsize;          /* initialized data size */

        long    bsize;          /* uninitialized data size */

        long    entry;          /* entry point */

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

        long    data_start      /* base of data for this file */

} AOUTHDR;
```

Figure 11-7: **aouthdr** Declaration

Section Headers

Every object file has a table of section headers to specify the layout of data within the file. The section header table consists of one entry for every section in the file. The information in the section header is described in Figure 11-8.

Bytes	Declaration	Name	Description
0-7	char	s_name	8-character null padded section name
8-11	long int	s_paddr	Physical address of section
12-15	long int	s_vaddr	Virtual address of section
16-19	long int	s_size	Section size in bytes
20-23	long int	s_scnptr	File pointer to raw data
24-27	long int	s_relptr	File pointer to relocation entries
28-31	long int	s_lnnoptr	File pointer to line number entries
32-33	unsigned short	s_nreloc	Number of relocation entries
34-35	unsigned short	s_nlnno	Number of line number entries
36-39	long int	s_flags	Flags (see Figure 11-9)

Figure 11-8: Section Header Contents

The size of a section is padded to a multiple of 4 bytes. File pointers are byte offsets that can be used to locate the start of data, relocation, or line number entries for the section. They can be readily used with the UNIX System function **fseek**(3S).

Flags

The lower 2 bytes of the flag field indicate a section type. The flags are described in Figure 11-9.

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

Figure 11-9: Section Header Flags

Section Header Declaration

The C structure declaration for the section headers is described in Figure 11-10. This declaration may be found in the header file **scnhdr.h**.

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

        long      s_relptr;       /* file ptr to relocation */

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

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

        unsigned short  s_nlnno;  /* number of line number entries */

        long      s_flags;        /* flags */

};

#define  SCNHDR   struct scnhdr
#define  SCNHSZ   sizeof(SCNHDR)
```

Figure 11-10: Section Header Declaration

.bss **Section Header**

The one deviation from the normal rule in the section header table is the entry for uninitialized data in a **.bss** section. A **.bss** section has a size and symbols that refer to it, and symbols that are defined in it. At the same time, a **.bss** section has no relocation entries, no line number entries, and no data. Therefore, a **.bss** section has an entry in the section header table but occupies no space elsewhere in the file. In this case, the number of relocation and line number entries, as well as all file pointers in a **.bss** section header, are 0. The same is true of the STYP_NOLOAD and STYP_DSECT sections.

Sections

Figure 11-1 shows that section headers are followed by the appropriate number of bytes of text or data. The raw data for each section begins on a 4-byte boundary in the file.

Link editor SECTIONS directives (see Chapter 12) allow users to do the following, among other things:

■ describe how input sections are to be combined

■ direct the placement of output sections

■ rename output sections.

If no SECTIONS directives are given, each input section appears in an output section of the same name. For example, if a number of object files, each with a **.text** section, are linked together, the output object file contains a single **.text** section made up of the combined input **.text** sections.

Relocation Information

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the format described in Figure 11-11.

Bytes	Declaration	Name	Description
0-3	**long int**	**r_vaddr**	(Virtual) address of reference
4-7	**long int**	**r_symndx**	Symbol table index
8-9	**unsigned short**	**r_type**	Relocation type

Figure 11-11: Relocation Section Contents

The first 4 bytes of the entry are the virtual address of the text or data to which this entry applies. The next field is the index, counted from 0, of the symbol table entry that is being referenced. The type field indicates the type of relocation to be applied.

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated. The currently recognized relocation types are given in Figure 11-12.

Mnemonic	Flag	Meaning
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_DIR16 *	01	Direct, 16-bit reference to a symbol's virtual address.
R_REL16 *	02	"PC-relative", 16-bit reference to a symbol's virtual address. Relative references occur in instructions such as jumps and calls.
R_DIR32	06	Direct 32-bit reference to the symbol's virtual address.
R_SEG12 *	011	Direct, 16-bit reference to the segment-selector bits of a 32-bit virtual address.
R_PCRLONG †	024	"PC_relative", 32-bit reference to a symbol's virtual address.

   * 80286 Computer only.
   † 80386 Computer only.

Figure 11-12: Relocation Types

Relocation Entry Declaration

The structure declaration for relocation entries is given in Figure 11-13. This declaration may be found in the header file **reloc.h**.

```
struct reloc
{
    long            r_vaddr;    /* virtual address of reference */

    long            r_symndx;   /* index into symbol table */

    unsigned short  r_type;     /* relocation type */
};

#define RELOC    struct reloc

#define RELSZ    10
```

Figure 11-13: Relocation Entry Declaration

Line Numbers

When invoked with the –g option, the cc and f77 commands cause an entry in the object file for every source line where a breakpoint can be inserted. You can then reference line numbers when using a software debugger like sdb. All line numbers in a section are grouped by function as shown in Figure 11-14.

symbol index	0
physical address	line number
physical address	line number
.	.
.	.
.	.
symbol index	0
physical address	line number
physical address	line number

Figure 11-14: Line Number Grouping

The first entry in a function grouping has line number 0 and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries are relative to the beginning of the function and appear in increasing order of address.

Line Number Declaration

The structure declaration currently used for line number entries is given in Figure 11-15.

```
struct lineno
{
        union
        {
                long    l_symndx;   /* symtbl index of func name */

                long    l_paddr;    /* paddr of line number */
        } l_addr;
        unsigned short  l_lnno;     /* line number */

};

#define LINENO      struct lineno
#define LINESZ      6
```

Figure 11-15: Line Number Entry Declaration

Symbol Table

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the sequence shown in Figure 11-16.

filename 1
function 1
local symbols for function 1
function 2
local symbols for function 2
. . .
statics
. . .
filename 2
function 1
local symbols for function 1
. . .
statics
. . .
defined global symbols
undefined global symbols

Figure 11-16: COFF Symbol Table

───

The word statics in Figure 11-16 means symbols defined with the C language storage class static outside any function. The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the value, the type, and other information.

Special Symbols

The symbol table contains some special symbols that are generated by **as** and other tools. These symbols are given in Figure 11-17.

Symbol	Meaning
.file	filename
.text	address of .text section
.data	address of .data section
.bss	address of .bss section
.bb	address of start of inner block
.eb	address of end of inner block
.bf	address of start of function
.ef	address of end of function
.target	pointer to the structure or union returned by a function
.xfake	dummy tag name for structure, union, or enumeration
.eos	end of members of structure, union, or enumeration
etext	next available address after the end of the output section .text
edata	next available address after the end of the output section .data
end	next available address after the end of the output section .bss

Figure 11-17: Special Symbols in the Symbol Table

Six of these special symbols occur in pairs. The **.bb** and **.eb** symbols indicate the boundaries of inner blocks; a **.bf** and **.ef** pair brackets each function. An *.xfake* and **.eos** pair names and defines the limit of structures, unions, and enumerations that were not named. The **.eos** symbol also appears after named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler invents a name to be used in the symbol table. The name chosen for the symbol table is *.xfake*, where x is an integer. If there are three unnamed structures, unions, or enumerations in the source, their tag names are **.0fake**, **.1fake**, and **.2fake**. Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entries.

Inner Blocks

The C language defines a block as a compound statement that begins and ends with braces, { and }. An inner block is a block that occurs within a function (which is also a block).

For each inner block that has local symbols defined, a special symbol **.bb** is put in the symbol table immediately before the first local symbol of that block. Also a special symbol **.eb** is put in the symbol table immediately after the last local symbol of that block. The sequence is shown in Figure 11-18.

.bb
local symbols
for that block
.eb

Figure 11-18: Special Symbols (**.bb** and **.eb**)

Because inner blocks can be nested by several levels, the **.bb**-**.eb** pairs and associated symbols may also be nested (see Figure 11-19).

```
{                              /* block 1 */
     int i;
     char c;
     ...
     {                         /* block 2 */
         long a;
         ...
       {                       /* block 3 */
              int x;
              ....
       }                       /* block 3 */

     }                         /* block 2 */

     {                         /* block 4 */
              long i;
              ...
     }                         /* block 4 */
}                              /* block 1 */
```

Figure 11-19: Nested blocks

The symbol table would look like Figure 11-20.

.bb for block 1
i
c
.bb for block 2
a
.bb for block 3
x
.eb for block 3
.eb for block 2
.bb for block 4
i
.eb for block 4
.eb for block 1

Figure 11-20: Example of the Symbol Table

Symbols and Functions

For each function, a special symbol **.bf** is put between the function name and the first local symbol of the function in the symbol table. Also, a special symbol **.ef** is put immediately after the last local symbol of the function in the symbol table. The sequence is shown in Figure 11-21.

function name
.bf
local symbol
.ef

Figure 11-21: Symbols for Functions

Symbol Table Entries

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries each contain 18 bytes of information. The meaning of each of the fields in the symbol table entry is described in Figure 11-22. It should be noted that indices for symbol table entries begin at 0 and count upward. Each auxiliary entry also counts as one symbol.

Bytes	Declaration	Name	Description
0-7	(see text below)	_n	These 8 bytes contain either a symbol name or an index to a symbol
8-11	long int	n_value	Symbol value; storage class dependent
12-13	short	n_scnum	Section number of symbol
14-15	unsigned short	n_type	Basic and derived type specification
16	char	n_sclass	Storage class of symbol
17	char	n_numaux	Number of auxiliary entries

Figure 11-22: Symbol Table Entry Format

Symbol Names

The first 8 bytes in the symbol table entry are a union of a character array and two longs. If the symbol name is eight characters or less, the (null-padded) symbol name is stored there. If the symbol name is longer than eight characters, then the entire symbol name is stored in the string table. In this case, the 8 bytes contain two long integers, the first is zero, and the second is the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first 4 bytes serve to distinguish a symbol table entry with an offset from one with a name in the first 8 bytes as shown in Figure 11-23.

Bytes	Declaration	Name	Description
0-7	char	n_name	8-character null-padded symbol name
0-3	long	n_zeroes	Zero in this field indicates the name is in the string table
4-7	long	n_offset	Offset of the name in the string table

Figure 11-23: Name Field

Special symbols generated by the C Compilation System are discussed above in "Special Symbols."

Storage Classes

The storage class field has one of the values described in Figure 11-24. These **#define**'s may be found in the header file **storclass.h**.

Mnemonic	Value	Storage Class
C_EFCN	−1	physical end of a function
C_NULL	0	–
C_AUTO	1	automatic variable
C_EXT	2	external symbol
C_STAT	3	static
C_REG	4	register variable
C_EXTDEF	5	external definition
C_LABEL	6	label
C_ULABEL	7	undefined label
C_MOS	8	member of structure
C_ARG	9	function argument
C_STRTAG	10	structure tag
C_MOU	11	member of union
C_UNTAG	12	union tag
C_TPDEF	13	type definition
C_USTATIC	14	uninitialized static
C_ENTAG	15	enumeration tag
C_MOE	16	member of enumeration
C_REGPARM	17	register parameter
C_FIELD	18	bit field
C_BLOCK	100	beginning and end of block
C_FCN	101	beginning and end of function
C_EOS	102	end of structure
C_FILE	103	file name
C_LINE	104	used only by utility programs
C_ALIAS	105	duplicated tag
C_HIDDEN	106	like static, used to avoid name conflicts

Figure 11-24: Storage Classes

All of these storage classes except for C_ALIAS and C_HIDDEN are generated by the **cc** or **as** commands. The compress utility, **cprs**, generates the C_ALIAS mnemonic. This utility (described in the *Programmer's Reference Manual*) removes duplicated structure, union, and enumeration definitions and puts alias entries in their places. The storage class C_HIDDEN is not used by any UNIX System tools.

Some of these storage classes are used only internally by the C Compilation Systems. These storage classes are C_EFCN, C_EXTDEF, C_ULABEL, C_USTATIC, and C_LINE.

Storage Classes for Special Symbols

Some special symbols are restricted to certain storage classes. They are given in Figure 11-25.

Special Symbol	Storage Class
.file	C_FILE
.bb	C_BLOCK
.eb	C_BLOCK
.bf	C_FCN
.ef	C_FCN
.target	C_AUTO
.xfake	C_STRTAG, C_UNTAG, C_ENTAG
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

Figure 11-25: Storage Class by Special Symbols

Also some storage classes are used only for certain special symbols. They are summarized in Figure 11-26.

Storage Class	Special Symbol
C_BLOCK	.bb, .eb
C_FCN	.bf, .ef
C_EOS	.eos
C_FILE	.file

Figure 11-26: Restricted Storage Classes

Symbol Value Field

The meaning of the value of a symbol depends on its storage class. This relationship is summarized in Figure 11-27.

Storage Class	Meaning of Value
C_AUTO	stack offset in bytes
C_EXT	relocatable address
C_STAT	relocatable address
C_REG	register number
C_LABEL	relocatable address
C_MOS	offset in bytes
C_ARG	stack offset in bytes
C_STRTAG	0
C_MOU	0
C_UNTAG	0
C_TPDEF	0
C_ENTAG	0
C_MOE	enumeration value
C_REGPARM	register number
C_FIELD	bit displacement
C_BLOCK	relocatable address
C_FCN	relocatable address
C_EOS	size
C_FILE	(see text below)
C_ALIAS	tag index
C_HIDDEN	relocatable address

Figure 11-27: Storage Class and Value

If a symbol has storage class C_FILE, the value of that symbol equals the symbol table entry index of the next **.file** symbol. That is, the **.file** entries form a one-way linked list in the symbol table. If there are no more **.file** entries in the symbol table, the value of the symbol is the index of the first global symbol.

Relocatable symbols have a value equal to the virtual address of that symbol. When the section is relocated by the link editor, the value of these symbols changes.

Section Number Field

Section numbers are listed in Figure 11-28.

Mnemonic	Section Number	Meaning
N_DEBUG	−2	Special symbolic debugging symbol
N_ABS	−1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1-077777	Section number where symbol is defined

Figure 11-28: Section Number

A special section number (−2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the name of the file. A section number of −1 indicates that the symbol has a value but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and **.eos** symbols.

With one exception, a section number of 0 indicates a relocatable external symbol that is not defined in the current file. The one exception is a multiply-defined external symbol (i.e., FORTRAN common or an uninitialized variable-defined external to a function in C). In the symbol table of each file where the symbol is defined, the section number of the symbol is 0, and the value of the symbol is a positive number giving the size of the symbol. When the files are combined to form an executable object file, the link editor combines all the input symbols of the same name into one symbol with the section number of the **.bss** section. The maximum size of all the input symbols with the same name is used to allocate space for the symbol and the value becomes the address of the symbol. This is the only case where a symbol has a section number of 0 and a non-zero value.

Section Numbers and Storage Classes

Symbols having certain storage classes are also restricted to certain section numbers. They are summarized in Figure 11-29.

Storage Class	Section Number
C_AUTO	N_ABS
C_EXT	N_ABS, N_UNDEF, N_SCNUM
C_STAT	N_SCNUM
C_REG	N_ABS
C_LABEL	N_UNDEF, N_SCNUM
C_MOS	N_ABS
C_ARG	N_ABS
C_STRTAG	N_DEBUG
C_MOU	N_ABS
C_UNTAG	N_DEBUG
C_TPDEF	N_DEBUG
C_ENTAG	N_DEBUG
C_MOE	N_ABS
C_REGPARM	N_ABS
C_FIELD	N_ABS
C_BLOCK	N_SCNUM
C_FCN	N_SCNUM
C_EOS	N_ABS
C_FILE	N_DEBUG
C_ALIAS	N_DEBUG

Figure 11-29: Section Number and Storage Class

Type Entry

The type field in the symbol table entry contains information about the basic and derived type for the symbol. This information is generated by the C Compilation System only if the **-g** option is used. Each symbol has exactly one basic or fundamental type but can have more than one derived type. The format of the 16-bit type entry is

d6	d5	d4	d3	d2	d1	typ
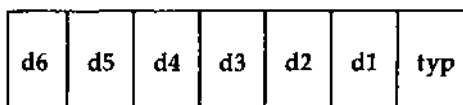
Bits 0 through 3, called **typ**, indicate one of the fundamental types given in Figure 11-30.

Mnemonic	Value	Type
T_NULL	0	type not assigned
T_ARG	1	Function argument (used only by compiler)
T_CHAR	2	character
T_SHORT	3	short integer
T_INT	4	integer
T_LONG	5	long integer
T_FLOAT	6	floating point
T_DOUBLE	7	double word
T_STRUCT	8	structure
T_UNION	9	union
T_ENUM	10	enumeration
T_MOE	11	member of enumeration
T_UCHAR	12	unsigned character
T_USHORT	13	unsigned short
T_UINT	14	unsigned integer
T_ULONG	15	unsigned long

Figure 11-30: Fundamental Types

Bits 4 through 15 are arranged as six 2-bit fields marked **d1** through **d6**. These **d** fields represent levels of the derived types given in Figure 11-31.

Mnemonic	Value	Type
DT_NON	0	no derived type
DT_PTR	1	pointer
DT_FCN	2	function
DT_ARY	3	array

Figure 11-31: Derived Types

The following examples demonstrate the interpretation of the symbol table entry representing type.

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

Here **func** is the name of a function that returns a pointer to a character. The fundamental type of **func** is 2 (character), the **d1** field is 2 (function), and the **d2** field is 1 (pointer). Therefore, the type word in the symbol table for **func** contains the hexadecimal number 0x62, which is interpreted to mean a function that returns a pointer to a character.

```
short *tabptr[10][25][3];
```

Here **tabptr** is a three-dimensional array of pointers to short integers. The fundamental type of **tabptr** is 3 (short integer); the **d1**, **d2**, and **d3** fields each contains a 3 (array), and the **d4** field is 1 (pointer). Therefore, the type entry in the symbol table contains the hexadecimal number 0x7f3 indicating a three-dimensional array of pointers to short integers.

Type Entries and Storage Classes

Figure 11-32 shows the type entries that are legal for each storage class.

Storage Class	d Entry			typ Entry Basic Type
	Function?	Array?	Pointer?	
C_AUTO	no	yes	yes	Any except T_MOE
C_EXT	yes	yes	yes	Any except T_MOE
C_STAT	yes	yes	yes	Any except T_MOE
C_REG	no	no	yes	Any except T_MOE
C_LABEL	no	no	no	T_NULL
C_MOS	no	yes	yes	Any except T_MOE
C_ARG	yes	no	yes	Any except T_MOE
C_STRTAG	no	no	no	T_STRUCT
C_MOU	no	yes	yes	Any except T_MOE
C_UNTAG	no	no	no	T_UNION
C_TPDEF	no	yes	yes	Any except T_MOE
C_ENTAG	no	no	no	T_ENUM
C_MOE	no	no	no	T_MOE
C_REGPARM	no	no	yes	Any except T_MOE
C_FIELD	no	no	no	T_ENUM, T_UCHAR, T_USHORT, T_UNIT, T_ULONG
C_BLOCK	no	no	no	T_NULL
C_FCN	no	no	no	T_NULL
C_EOS	no	no	no	T_NULL
C_FILE	no	no	no	T_NULL
C_ALIAS	no	no	no	T_STRUCT, T_UNION, T_ENUM

Figure 11-32: Type Entries by Storage Class

Conditions for the **d** entries apply to **d1** through **d6**, except that it is impossible to have two consecutive derived types of function.

Although function arguments can be declared as arrays, they are changed to pointers by default. Therefore, no function argument can have array as its first derived type.

Structure for Symbol Table Entries

The C language structure declaration for the symbol table entry is given in Figure 11-33. This declaration may be found in the header file **syms.h**.

```
struct syment
{
    union
    {
        char            _n_name[SYMNMLEN];    /* symbol name*/
        struct
        {
            long    _n_zeroes;    /* symbol name */

            long    _n_offset;    /* location in string table */
        } _n_n;
        char            *_n_nptr[2];  /* allows overlaying */
    } _n;
    unsigned long    n_value;              /* value of symbol */

    short            n_scnum;             /* section number */

    unsigned short   n_type;              /* type and derived */

    char             n_sclass;            /* storage class */

    char             n_numaux;            /* number of aux entries */
};

#define n_name      _n._n_name
#define n_zeroes    _n._n_n._n_zeroes
#define n_offset    _n._n_n._n_offset
#define n_nptr      _n._n_nptr[1]

#define SYMNMLEN  8
#define SYMESZ    18   /* size of a symbol table entry */
```

Figure 11-33: Symbol Table Entry Declaration

Auxiliary Table Entries

An auxiliary table entry of a symbol contains the same number of bytes as the symbol table entry. However, unlike symbol table entries, the format of an auxiliary table entry of a symbol depends on its type and storage class. They are summarized in Figure 11-34.

Name	Storage Class	Type Entry		Auxiliary Entry Format
		d1	typ	
.file	C_FILE	DT_NON	T_NULL	file name
.text,.data, .bss	C_STAT	DT_NON	T_NULL	section
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	tag name
.eos	C_EOS	DT_NON	T_NULL	end of structure
fcname	C_EXT C_STAT	DT_FCN	(Note 1)	function
arrname	(Note 2)	DT_ARY	(Note 1)	array
.bb,.eb	C_BLOCK	DT_NON	T_NULL	beginning and end of block
.bf,.ef	C_FCN	DT_NON	T_NULL	beginning and end of function
name related to structure, union, enumeration	(Note 2)	DT_PTR, DT_ARR, DT_NON	T_STRUCT, T_UNION, T_ENUM	name related to structure, union, enumeration

Figure 11-34: Auxiliary Symbol Table Entries

Notes to Figure 11-34:
1. Any except T_MOE.
2. C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF.

In Figure 11-34, *tagname* means any symbol name including the special symbol *.xfake*, and *fcname* and *arrname* represent any symbol name for a function or an array respectively. Any symbol that satisfies more than one condition in Figure 11-34 should have a union format in its auxiliary entry.

NOTE	It is a mistake to assume how many auxiliary entries are associated with any given symbol table entry. This information is available and should be obtained from the **n_numaux** field in the symbol table.

File Names

Each of the auxiliary table entries for a file name contains a 14-character file name in bytes 0 through 13. The remaining bytes are 0.

Sections

The auxiliary table entries for sections have the format as shown in Figure 11-35.

Bytes	Declaration	Name	Description
0-3	long int	x_scnlen	section length
4-5	unsigned short	x_nreloc	number of relocation entries
6-7	unsigned short	x_nlinno	number of line numbers
8-17	–	–	unused (filled with zeroes)

Figure 11-35: Format for Auxiliary Table Entries for Sections

Tag Names

The auxiliary table entries for tag names have the format shown in Figure 11-36.

Bytes	Declaration	Name	Description
0-5	–	–	unused (filled with zeroes)
6-7	unsigned short	x_size	size of structure, union, and enumeration
8-11	–	–	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry beyond this structure, union, or enumeration
16-17	–	–	unused (filled with zeroes)

Figure 11-36: Tag Names Table Entries

End of Structures

The auxiliary table entries for the end of structures have the format shown in Figure 11-37:

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	–	–	unused (filled with zeroes)
6-7	unsigned short	x_size	size of structure, union, or enumeration
8-17	–	–	unused (filled with zeroes)

Figure 11-37: Table Entries for End of Structures

Functions

The auxiliary table entries for functions have the format shown in Figure 11-38:

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-7	long int	x_fsize	size of function (in bytes)
8-11	long int	x_lnnoptr	file pointer to line number
12-15	long int	x_endndx	index of next entry beyond this point
16-17	unsigned short	x_tvndx	index of function's address in the transfer vector table (not used in the UNIX System)

Figure 11-38: Table Entries for Functions

Arrays

The auxiliary table entries for arrays have the format shown in Figure 11-39. Defining arrays having more than four dimensions produces a warning message.

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	unsigned short	x_lnno	line number of declaration
6-7	unsigned short	x_size	size of array
8-9	unsigned short	x_dimen[0]	first dimension
10-11	unsigned short	x_dimen[1]	second dimension
12-13	unsigned short	x_dimen[2]	third dimension
14-15	unsigned short	x_dimen[3]	fourth dimension
16-17	–	–	unused (filled with zeroes)

Figure 11-39: Table Entries for Arrays

End of Blocks and Functions

The auxiliary table entries for the end of blocks and functions have the format shown in Figure 11-40:

Bytes	Declaration	Name	Description
0-3	–	–	unused (filled with zeroes)
4-5	unsigned short	x_lnno	C-source line number
6-17	–	–	unused (filled with zeroes)

Figure 11-40: End of Block and Function Entries

Beginning of Blocks and Functions

The auxiliary table entries for the beginning of blocks and functions have the format shown in Figure 11-41:

Bytes	Declaration	Name	Description
0-3	–	–	unused (filled with zeroes)
4-5	unsigned short	x_lnno	C-source line number
6-11	–	–	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry past this block
16-17	–	–	unused (filled with zeroes)

Figure 11-41: Format for Beginning of Block and Function

Names Related to Structures, Unions, and Enumerations

The auxiliary table entries for structure, union, and enumeration symbols have the format shown in Figure 11-42:

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	–	–	unused (filled with zeroes)
6-7	unsigned short	x_size	size of the structure, union, or enumeration
8-17	–	–	unused (filled with zeroes)

Figure 11-42: Entries for Structures, Unions, and Enumerations

Aggregates defined by **typedef** may or may not have auxiliary table entries. For example,

```
typedef struct people STUDENT;

struct people
{
        char name[20];
        long id;
};

typedef struct people EMPLOYEE;
```

The symbol EMPLOYEE has an auxiliary table entry in the symbol table, but symbol STUDENT will not because it is a forward reference to a structure.

Auxiliary Entry Declaration

The C language structure declaration for an auxiliary symbol table entry is given in Figure 11-43. This declaration may be found in the header file **syms.h**.

```
union auxent
{
      struct
      {
            long   x_tagndx;
            union
            {
                  struct
                  {
                        unsigned short   x_lnno;
                        unsigned short   x_size;
                  } x_lnsz;
                  long   x_fsize;
            } x_misc;
            union
            {
                  struct
                          .
                          .
                          .
```

Figure 11-43: Auxiliary Symbol Table Entry (Sheet 1 of 2)

```
        .
        .
        .
                {
                        long    x_lnnoptr;
                        long    x_endndx;
                } x_fcn;
                struct
                {
                        unsigned short   x_dimen[DIMNUM];
                } x_ary;
            } x_fcnary;
            unsigned short   x_tvndx;
        } x_sym;
        struct
        {
            char   x_fname[FILNMLEN];
        } x_file;
        struct
        {
            long   x_scnlen;
            unsigned short   x_nreloc;
            unsigned short   x_nlinno;
        } x_scn;
        struct
        {
            long   x_tvfill;
            unsigned short   x_tvlen;
            unsigned short   x_tvran[2];
        } x_tv;
}
#define FILNMLEN   14
#define DIMNUM     4
#define AUXENT     union auxent
#define AUXESZ     18
```

Figure 11-43: Auxiliary Symbol Table Entry (Sheet 2 of 2)

String Table

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes; offsets into the string table, therefore, are greater than or equal to 4. For example, given a file containing two symbols (with names longer then eight characters, **long—name—1** and **another—one**) the string table has the format as shown in Figure 11-44:

'l'	'o'	'n'	'g'
'—'	'n'	'a'	'm'
'e'	'—'	'l'	'\0'
'a'	'n'	'o'	't'
'h'	'e'	'r'	'—'
'o'	'n'	'e'	'\0'

Figure 11-44: String Table

The index of **long—name—1** in the string table is 4 and the index of **another—one** is 16.

Access Routines

UNIX System V/386 Release 3.2 contains a set of access routines that are used for reading the various parts of a common object file. Although the calling program must know the detailed structure of the parts of the object file it processes, the routines effectively insulate the calling program from the

knowledge of the overall structure of the object file.

The access routines can be divided into four categories:

1. functions that open or close an object file
2. functions that read header or symbol table information
3. functions that position an object file at the start of a particular section of the object file
4. a function that returns the symbol table index for a particular symbol

These routines can be found in the library **libld.a** and are listed in Section 3 of the *Programmer's Reference Manual*. A summary of what is available can be found in the *Programmer's Reference Manual* under **ldfcn(4)**.

12 The Link Editor

The Link Editor

In Chapter 2 there was a discussion of link editor command line options, some of which may also be provided on the **cc**(1) command line. The commands **cc** and **ld** referred to in this section are documented in the *Programmer's Reference Manual*. This chapter contains information on the Link Editor Command Language. The command language enables you to

- specify the memory configuration of the target machine

- combine the sections of an object file in arrangements other than the default

- bind sections to specific addresses or within specific portions of memory

- define or redefine global symbols

Under most normal circumstances there is no compelling need to have such tight control over object files and where they are located in memory. When you do need to be very precise in controlling the link editor output, you do it by means of the command language.

Link editor command language directives are passed in a file named on the **ld**(1) command line. Any file named on the command line that is not identifiable as an object module or an archive library is assumed to contain directives. The following paragraphs define terms and describe conditions with which you need to be familiar before you begin to use the command language.

Memory Configuration

The virtual memory of the target machine is, for purposes of allocation, partitioned into configured and unconfigured memory. The default condition is to treat all memory as configured. It is common with microprocessor applications, however, to have different types of memory at different addresses. For example, an application might have 3K of PROM (Programmable Read-Only Memory) beginning at address 0, and 8K of ROM (Read-Only Memory) starting at 20K. Addresses in the range 3K to 20K–1 are then not configured. Unconfigured memory is treated as reserved or unusable by **ld**(1). Nothing can ever be linked into unconfigured memory. Thus, specifying a certain memory range to be unconfigured is one way of marking the addresses (in that range) illegal or nonexistent with respect to the linking process. Memory

configurations other than the default must be explicitly specified by you (the user).

Unless otherwise specified, all discussion in this document of memory, addresses, etc., are with respect to the configured sections of the address space.

Sections

A section of an object file is the smallest unit of relocation and must be a contiguous block of memory. A section is identified by a starting address and a size. Information describing all the sections in a file is stored in section headers at the start of the file. Sections from input files are combined to form output sections that contain executable text, data, or a mixture of both. Although there may be holes or gaps between input sections and between output sections, storage is allocated contiguously within each output section and may not overlap a hole in memory.

Addresses

The physical address of a section or symbol is the relative offset from address zero of the address space. The physical address of an object is not necessarily the location at which it is placed when the process is executed. For example, on a system with paging, the address is with respect to address zero of the virtual space, and the system performs another address translation.

Binding

It is often necessary to have a section begin at a specific, predefined address in the address space. The process of specifying this starting address is called binding, and the section in question is said to be bound to or bound at the required address. While binding is most commonly relevant to output sections, it is also possible to bind special absolute global symbols with an assignment statement in the **ld**(1) command language.

Object File

Object files are produced both by the assembler (typically as a result of calling the compiler) and by ld(1). ld(1) accepts relocatable object files as input and produces an output object file that may or may not be relocatable. Under certain special circumstances, the input object files given to ld(1) can also be absolute files.

Files produced from the compilation system may contain, among others, sections called .text and .data. The .text section contains the instruction text (executable instructions); .data contains initialized data variables. For example, if a C program contained the global (i.e., not inside a function) declaration

```
int i = 100;
```

and the assignment

```
i = 0;
```

then compiled code from the C assignment is stored in .text, and the variable i is located in .data.

Link Editor Command Language

Expressions

Expressions may contain global symbols, constants, and most of the basic C language operators. (See Figure 12-2, "Syntax Diagram for Input Directives.") Constants are as in C with a number recognized as decimal unless preceded with 0 for octal or 0x for hexadecimal. All numbers are treated as long integers. Symbol names may contain uppercase or lowercase letters, digits, and the underscore, _. Symbols within an expression have the value of the address of the symbol only. ld(1) does not do symbol table lookup to find the contents of a symbol, the dimensionality of an array, structure elements declared in a C program, etc.

ld(1) uses a lex-generated input scanner to identify symbols, numbers, operators, etc. The current scanner design makes the following names reserved and unavailable as symbol names or section names:

ADDR	BLOCK	GROUP	NEXT	RANGE	SPARE
ALIGN	COMMON	INFO	NOLOAD	REGIONS	PHY
ASSIGN	COPY	LENGTH	ORIGIN	SECTIONS	TV
BIND	DSECT	MEMORY	OVERLAY	SIZEOF	

addr	block	length	origin	sizeof
align	group	next	phy	spare
assign	l	o	range	
bind	len	org	s	

The operators that are supported, in order of precedence from high to low, are shown in Figure 12-1:

symbol
! ~ – (UNARY Minus)
* / %
+ – (BINARY Minus)
>> <<
== != > < <= >=
&
|
&&
||
= += –= *= /=

Figure 12-1: Operator Symbols

The above operators have the same meaning as in the C language. Operators on the same line have the same precedence.

Assignment Statements

External symbols may be defined and assigned addresses by means of the assignment statement. The syntax of the assignment statement is

        symbol = expression;

or

        symbol op= expression;

where *op* is one of the operators +, –, *, or / . Assignment statements must be terminated by a semicolon.

All assignment statements (with the exception of the one case described in the following paragraph) are evaluated after allocation has been performed. This occurs after all input-file-defined symbols are appropriately relocated but before the actual relocation of the text and data itself. Therefore, if an assignment statement expression contains any symbol name, the address used for that symbol in the evaluation of the expression reflects the symbol address in the output object file. References within text and data (to symbols given a value through an assignment statement) access this latest assigned value.

Assignment statements are processed in the same order in which they are input to **ld**(1).

Assignment statements are normally placed outside the scope of section definition directives (see "Section Definition Directives" under "Link Editor Command Language"). However, there exists a special symbol, called **dot**, (.), that can occur only within a section definition directive. This symbol refers to the current address of **ld**(1)'s location counter. Thus, assignment expressions involving . are evaluated during the allocation phase of **ld**(1). Assigning a value to the . symbol within a section definition directive can increment (but not decrement) **ld**(1)'s location counter and can create holes within the section, as described in "Section Definition Directives." Assigning the value of the . symbol to a conventional symbol permits the final allocated address (of a particular point within the link edit run) to be saved.

align is provided as a shorthand notation to allow alignment of a symbol to an n-byte boundary within an output section, where n is a power of 2. For example, the expression

    align(n)

is equivalent to

    $(. + n - 1) \mathbin{\&} {\sim}(n - 1)$

SIZEOF and ADDR are pseudo-functions that, given the name of a section, return the size or address of the section respectively. They may be used in symbol definitions outside of section directives.

Link editor expressions may have either an absolute or a relocatable value. When **ld**(1) creates a symbol through an assignment statement, the symbol's value takes on that type of expression. That type depends on the following rules:

■ An expression with a single relocatable symbol (and zero or more constants or absolute symbols) is relocatable.

■ The difference of two relocatable symbols from the same section is absolute.

■ All other expressions are combinations of the above.

Specifying a Memory Configuration

MEMORY directives are used to specify

■ the total size of the virtual space of the target machine

■ the configured and unconfigured areas of the virtual space

If no directives are supplied, **ld**(1) assumes that all memory is configured. The size of the default memory is dependent upon the target machine.

By means of MEMORY directives, an arbitrary name of up to eight characters is assigned to a virtual address range. Output sections can then be forced to be bound to virtual addresses within specifically named memory areas. Memory names may contain uppercase or lowercase letters, digits, and the special characters **$**, **.**, or **_**. Names of memory ranges are used by **ld**(1) only and are not carried in the output file symbol table or headers.

When MEMORY directives are used, all virtual memory not described in a MEMORY directive is considered to be unconfigured. Unconfigured memory is not used in **ld**(1)'s allocation process; hence nothing except DSECT sections can be link edited or bound to an address within unconfigured memory.

As an option on the MEMORY directive, attributes may be associated with a named memory area. In future releases this may be used to provide error checking. Currently, error checking of this type is not implemented.

The attributes currently accepted are

■ R : readable memory

■ W : writable memory

■ X : executable, i.e., instructions may reside in this memory

■ I : initializable, i.e., stack areas are typically not initialized

Other attributes may be added in the future if necessary. If no attributes are specified on a MEMORY directive or if no MEMORY directives are supplied, memory areas assume the attributes of R, W, X, and I.

The syntax of the MEMORY directive is as follows:

```
MEMORY
{
        name1 (attr) :      origin = n1, length = n2
        name2 (attr) :      origin = n3, length = n4
        etc.
}
```

The keyword **origin** (or **org** or **o**) must precede the origin of a memory range, and **length** (or **len** or **l**) must precede the length as shown in the above prototype. The **origin** operand refers to the virtual address of the memory range. **origin** and **length** are entered as long integer constants in either decimal, octal, or hexadecimal (standard C syntax). **origin** and **length** specifications, as well as individual MEMORY directives, may be separated by white space or a comma.

By specifying MEMORY directives, **ld**(1) can be told that memory is configured in some manner other than the default. For example, if it is necessary to prevent anything from being linked to the first 0x10000 words of memory, a MEMORY directive can accomplish this.

Section Definition Directives

The purpose of the SECTIONS directive is to describe how input sections are to be combined, to direct where to place output sections (both in relation to each other and to the entire virtual memory space), and to permit the renaming of output sections.

In the default case where no SECTIONS directives are given, all input sections of the same name appear in an output section of that name. If two object files are linked, one containing sections s1 and s2 and the other containing sections s3 and s4, the output object file contains the four sections s1, s2, s3, and s4. The order of these sections would depend on the order in which the link editor sees the input files.

The basic syntax of the SECTIONS directive is

```
SECTIONS
{
        secname1 ;
        {
                file_specifications,
                assignment_statements
        }
        secname2 :
        {
                file_specifications,
                assignment_statements
        }
etc.
}
```

The various types of section definition directives are discussed in the remainder of this section.

File Specifications

Within a section definition, the files and sections of files to be included in the output section are listed in the order in which they are to appear in the output section. Sections from an input file are specified by

        filename (secname)

or

        filename (secnam1 secnam2 . . .)

Sections of an input file are separated either by white space or commas, as are the file specifications themselves.

        filename [COMMON]

may be used in the same way to refer to all the uninitialized, unallocated global symbols in a file.

If a file name appears with no sections listed, then all sections from the file (but not the uninitialized, unallocated globals) are linked into the current output section. For example,

```
SECTIONS
{
        outsec1:
        {
                file1.o (sec1)
                file2.o
                file3.o (sec1, sec2)
        }
}
```

According to this directive, the order in which the input sections appear in the output section **outsec1** would be

- section **sec1** from file **file1.o**

- all sections from **file2.o**, in the order they appear in the file

- section **sec1** from file **file3.o**, and then section **sec2** from file **file3.o**

If there are any additional input files that contain input sections also named **outsec1**, these sections are linked following the last section named in the definition of **outsec1**. If there are any other input sections in **file1.o** or **file3.o**, they will be placed in output sections with the same names as the input sections unless they are included in other file specifications.

The code

```
*(secname)
```

may be used to indicate all previously unallocated input sections of the given name, regardless of what input file they are contained in.

Load a Section at a Specified Address

Bonding of an output section to a specific virtual address is accomplished by an ld(1) option as shown in the following SECTIONS directive example:

```
SECTIONS
{
        outsec addr:
        {
                . . .
        }
        etc.
}
```

The *addr* is the bonding address expressed as a C constant. If **outsec** does not fit at *addr* (perhaps because of holes in the memory configuration or because **outsec** is too large to fit without overlapping some other output section), **ld**(1) issues an appropriate error message. *addr* may also be the word BIND, followed by a parenthesized expression. The expression may use the pseudo-functions SIZEOF, ADDR, or NEXT. NEXT accepts a constant and returns the first multiple of that value that falls into configured unallocated memory; SIZEOF and ADDR accept previously defined sections.

As long as output sections do not overlap and there is enough space, they can be bound anywhere in configured memory. The SECTIONS directives defining output sections need not be given to **ld**(1) in any particular order, unless SIZEOF or ADDR is used.

The **ld**(1) does not ensure that the size of each section consists of an even number of bytes or that each section starts on an even byte boundary. The assembler ensures that the size (in bytes) of a section is evenly divisible by 4.

The **ld**(1) directives can be used to force a section to start on an odd byte boundary, although this is not recommended. If a section starts on an odd byte boundary, the section's contents are either accessed incorrectly or are not executed properly. When a user specifies an odd byte boundary, **ld**(1) issues a warning message.

Aligning an Output Section

It is possible to request that an output section be bound to a virtual address that falls on an *n*-byte boundary, where *n* is a power of 2. The ALIGN option of the SECTIONS directive performs this function, so that the option

        ALIGN(n)

is equivalent to specifying a bonding address of

        (. + n − 1) &~(n − 1)

For example

```
        SECTIONS
        {
                outsec  ALIGN(0x20000) :
                {
                        . . .
                }
                etc.
        }
```

The output section **outsec** is not bound to any given address but is placed at some virtual address that is a multiple of 0x20000 (e.g., at address 0x0, 0x20000, 0x40000, 0x60000, etc.).

Grouping Sections Together

The default allocation algorithm for **ld**(1) does the following:

- It links all input **.init** sections together, followed by **.text** sections, into one output section. This output section is called **.text** and is bound to an address of 0x0 plus the size of all headers in the output file.

- It links all input **.data** sections together into one output section. This output section is called **.data** and, in paging systems, is bound to an address aligned to a machine-dependent constant plus a number dependent on the size of headers and text.

■ It links all input **.bss** sections together with all uninitialized, unallocated global symbols, into one output section. This output section is called **.bss** and is allocated so as to immediately follow the output section **.data**.

Specifying any SECTIONS directives results in this default allocation not being performed. Rather than relying on the **ld**(1) default algorithm, if you are manipulating COFF files, the one certain way of determining address and order information is to take it from the file and section headers. The default allocation of **ld**(1) is equivalent to supplying the following directive, where *align_value* is a machine-dependent constant.

```
SECTIONS
{
        .text sizeof_headers : { *(.init) *(.text) *(.fini)}
        GROUP BIND( NEXT(align_value) +
                ((SIZEOF(.text) + ADDR(.text)) % 0x2000)) ;
        {
                .data   : { }
                .bss    : { }
        }
}
```

The GROUP command ensures that the two output sections, **.data** and **.bss**, are allocated (e.g., grouped) together. Bonding or alignment information is supplied only for the group and not for the output sections contained within the group. The sections making up the group are allocated in the order listed in the directive.

For compatibility with UNIX System V Release 2, these addresses cannot change. Unfortunately, **.init** sections in the algorithm above will interfere with the placement of the signal recovery routines. Hence the **.text** sections are linked into the **a.out .text** section first. The **.init** sections (for shared libraries) and the **.fini** sections follow all of the **.text** sections. Routines in **crt1.0** branch to the **.init** sections before calling the **main**() function of the program.

If **.text**, **.data**, and **.bss** are to be placed in the same segment, the following SECTIONS directive is used:

```
SECTIONS
{
        GROUP           :
        {
                .text   : { }
                .data   : { }
                .bss    : { }
        }
}
```

Note that there are still three output sections (**.text**, **.data**, and **.bss**), but now they are allocated into consecutive virtual memory.

This entire group of output sections could be bound to a starting address or aligned simply by adding a field to the GROUP directive. To bind to 0xC0000, use

```
        GROUP 0xC0000 : {
```

To align to 0x10000, use

```
        GROUP ALIGN(0x10000) : {
```

With this addition, first the output section **.text** is bound at 0xC0000 (or is aligned to 0x10000); then the remaining members of the group are allocated in order of their appearance into the next available memory locations.

When the GROUP directive is not used, each output section is treated as an independent entity:

```
SECTIONS
{
        .text   : { }
        .data ALIGN(0x400000)   : { }
        .bss    : { }
}
```

The **.text** section starts at virtual address 0x0 (if it is in configured memory) and the **.data** section at a virtual address aligned to 0x400000. The **.bss** section follows immediately after the **.text** section if there is enough space. If there is not, it follows the **.data** section. The order in which output sections are defined to **ld**(1) cannot be used to force a certain allocation order in the output file.

Creating Holes Within Output Sections

The special symbol dot, (.), appears only within section definitions and assignment statements. When it appears on the left side of an assignment statement, . causes **ld**(1)'s location counter to be incremented or reset and a hole left in the output section. Holes built into output sections in this manner take up physical space in the output file and are initialized using a fill character, either the default fill character (0x00) or a supplied fill character. See the discussion of filling holes in "Initialized Section Holes or **.bss** Sections" in this chapter.

Consider the following section definition:

```
outsec:
{
        . += 0x1000;
        f1.o (.text)
        . += 0x100;
        f2.o (.text)
        . = align (4);
        f3.o (.text)
}
```

The effect of this command is as follows:

■ A 0x1000 byte hole, filled with the default fill character, is left at the beginning of the section. Input section **f1.o (.text)** is linked after this hole.

■ The **.text** section of input file **f2.o** begins at 0x100 bytes following the end of **f1.o (.text)**.

■ The **.text** section of **f3.o** is linked to start at the next full word boundary following the **.text** section of **f2.o** with respect to the beginning of **outsec**.

For the purposes of allocating and aligning addresses within an output section, **ld**(1) treats the output section as if it began at address zero. As a result, if, in the above example, **outsec** ultimately is linked to start at an odd address, then the part of **outsec** built from **f3.o (.text)** also starts at an odd address—even though **f3.o (.text)** is aligned to a full word boundary. This is prevented by specifying an alignment factor for the entire output section.

```
outsec ALIGN(4) : {
```

Expressions that decrement . are illegal. For example, subtracting a value from the location counter is not allowed, since overwrites are not allowed. The most common operators in expressions that assign a value to . are += and **align**.

Creating and Defining Symbols at Link-Edit Time

The assignment instruction of ld(1) can be used to give symbols a value that is link-edit dependent. Typically, there are three types of assignments:

1. use of . to adjust ld(1)'s location counter during allocation

2. use of . to assign an allocation-dependent value to a symbol

3. assigning an allocation-independent value to a symbol

Case 1 has already been discussed in the previous section.

Case 2 provides a means to assign addresses (known only after allocation) to symbols. For example,

```
SECTIONS
{
        outsc1: {...}
        outsc2:
        {
                file1.o (s1)
                s2_start = . ;
                file2.o (s2)
                s2_end = . - 1;
        }
}
```

The symbol s2_start is defined to be the address of file2.o(s2), and s2_end is the address of the last byte of file2.o(s2).

Consider the following example:

```
SECTIONS
{
        outsc1:
        {
                file1.o (.data)
                mark = .;
                . += 4;
                file2.o (.data)
        }
}
```

In this example, the symbol **mark** is created and is equal to the address of the first byte beyond the end of **file1.o**'s **.data** section. Four bytes are reserved for a future run-time initialization of the symbol **mark**. The type of the symbol is a long integer (32 bits).

Assignment instructions involving . must appear within SECTIONS definitions since they are evaluated during allocation. Assignment instructions that do not involve . can appear within SECTIONS definitions but typically do not. Such instructions are evaluated after allocation is complete. Reassignment of a defined symbol to a different address is dangerous. For example, if a symbol within **.data** is defined, initialized, and referenced within a set of object files being link-edited, the symbol table entry for that symbol is changed to reflect the new, reassigned physical address. However, the associated initialized data is not moved to the new address, and there may be references to the old address. The **ld**(1) issues warning messages for each defined symbol that is being redefined within an ifile. However, assignments of absolute values to new symbols are safe because there are no references or initialized data associated with the symbol.

Allocating a Section Into Named Memory

It is possible to specify that a section be linked (somewhere) within a specific named memory (as previously specified on a MEMORY directive). (The > notation is borrowed from the UNIX System concept of redirected output.) For example,

```
MEMORY
{
        mem1:           o=0x000000      l=0x10000
        mem2 (RW):      o=0x020000      l=0x40000
        mem3 (RW):      o=0x070000      l=0x40000
        mem1:           o=0x120000      l=0x04000
}

SECTIONS
{
        outsec1: { f1.o(.data) } > mem1
        outsec2: { f2.o(.data) } > mem3
}
```

This directs **ld**(1) to place **outsec1** anywhere within the memory area named **mem1** (i.e., somewhere within the address range 0x0-0xFFFF or 0x120000-0x123FFF). The **outsec2** is to be placed somewhere in the address range 0x70000-0xAFFFF.

Initialized Section Holes or .bss Sections

When holes are created within a section (as in the example in "Creating Holes within Output Sections"), **ld**(1) normally puts out bytes of zero as fill. By default, **.bss** sections are not initialized at all; that is, no initialized data is generated for any **.bss** section by the assembler nor supplied by the link editor, not even zeros.

Initialization options can be used in a SECTIONS directive to set such holes or output **.bss** sections to an arbitrary 2-byte pattern. Such initialization options apply only to **.bss** sections or holes. As an example, an application might want an uninitialized data table to be initialized to a constant value without recompiling the **.o** file or a hole in the text area to be filled with a transfer to an error routine.

Either specific areas within an output section or the entire output section may be specified as being initialized. However, since no text is generated for an uninitialized **.bss** section, if part of such a section is initialized, then the entire section is initialized. In other words, if a **.bss** section is to be combined with a **.text** or **.data** section (both of which are initialized) or if part of an output **.bss** section is to be initialized, then one of the following will apply:

■ Explicit initialization options must be used to initialize all **.bss** sections in the output section.

■ **ld**(1) will use the default fill value to initialize all **.bss** sections in the output section.

Consider the following **ld**(1) file:

```
         SECTIONS
         {
                 sec1:
                 {
                         f1.o
                         . =+ 0x200;
                         f2.o (.text)
                 } = 0xDFFF
                 sec2:
                 {
                         f1.o (.bss)
                         f2.o (.bss) = 0x1234
                 }
                 sec3:
                 {
                         f3.o (.bss)
                         . . .
                 } = 0xFFFF
                  sec4: { f4.o (.bss) }
         }
```

In the example above, the 0x200 byte hole in section **sec1** is filled with the value 0xDFFF. In section **sec2**, **f1.o(.bss)** is initialized to the default fill value of 0x00, and **f2.o(.bss)** is initialized to 0x1234. All **.bss** sections within sec3 as well as all holes are initialized to 0xFFFF. Section **sec4** is not initialized; that is, no data is written to the object file for this section.

Notes and Special Considerations

Changing the Entry Point

The UNIX System **a.out** optional header contains a field for the (primary) entry point of the file. This field is set using one of the following rules (listed in the order they are applied):

The value of the symbol specified with the −e option, if present, is used.

1. The value of the symbol **_start**, if present, is used.

2. The value of the symbol **main**, if present, is used.

3. The value zero is used.

Thus, an explicit entry point can be assigned to this **a.out** header field through the −e option or by using an assignment instruction in an ifile of the form

```
_start   =   expression;
```

If **ld**(1) is called through **cc**(1), a startup routine is automatically linked in. Then, when the program is executed, the routine **exit**(2) is called after the main routine finishes to close file descriptors and do other cleanup. The user must therefore be careful when calling **ld**(1) directly or when changing the entry point. The user must supply the start-up routine or make sure that the program always calls exit rather than falling through the end. Otherwise, the program will dump core.

Use of Archive Libraries

Each member of an archive library (e.g., **libc.a**) is a complete object file. Archive libraries are created with the **ar**(1) command from object files generated by **cc** or **as**. **ar**(1) is documented in the *Programmer's Reference Manual*. An archive library is always processed using selective inclusion: only those members that resolve existing undefined-symbol references are taken from the library for link editing. Libraries can be placed both inside and outside section definitions. In both cases, a member of a library is included for linking whenever

■ there exists a reference to a symbol defined in that member

■ the reference is found by **ld**(1) prior to the actual scanning of the library

When a library member is included by searching the library inside a SEC-TIONS directive, all input sections from the library member are included in the output section being defined. When a library member is included by searching the library outside of a SECTIONS directive, all input sections from the library member are included into the output section with the same name. If necessary, new output sections are defined to provide a place to put the input sections. Note, however, that

■ specific members of a library cannot be referenced explicitly in an ifile

■ the default rules for the placement of members and sections cannot be overridden when they apply to archive library members

The −l option is a shorthand notation for specifying an input file coming from a predefined set of directories and having a predefined name. By convention, such files are archive libraries. However, they need not be so. Furthermore, archive libraries can be specified without using the −l option by simply giving the (full or relative) UNIX System file path.

The ordering of archive libraries is important since for a member to be extracted from the library it must satisfy a reference that is known to be unresolved at the time the library is searched. Archive libraries can be specified more than once. They are searched every time they are encountered. Archive files have a symbol table at the beginning of the archive. **ld**(1) will cycle through this symbol table until it has determined that it cannot resolve any more references from that library.

Consider the following example:

■ The input files **file1.o** and **file2.o** each contain a reference to the external function FCN.

■ Input **file1.o** contains a reference to symbol ABC.

■ Input **file2.o** contains a reference to symbol XYZ.

■ Library **liba.a**, member 0, contains a definition of XYZ.

■ Library **libc.a**, member 0, contains a definition of ABC.

■ Both libraries have a member 1 that defines FCN.

If the **ld**(1) command were entered as

> **ld file1.o –la file2.o –lc**

then the FCN references are satisfied by **liba.a**, member 1; ABC is obtained from **libc.a**, member 0; and XYZ remains undefined (because the library **liba.a** is searched before **file2.o** is specified). If the **ld**(1) command were entered as

> **ld file1.o file2.o –la –lc**

then the FCN references are satisfied by **liba.a**, member 1; ABC is obtained from **libc.a**, member 0; and XYZ is obtained from **liba.a**, member 0. If the **ld**(1) command were entered as

> **ld file1.o file2.o –lc –la**

then the FCN references are satisfied by **libc.a**, member 1; ABC is obtained from **libc.a**, member 0; and XYZ is obtained from **liba.a**, member 0.

The **–u** option is used to force the linking of library members when the link edit run does not contain an actual external reference to the members. For example,

> **ld –u rout1 –la**

creates an undefined symbol called **rout1** in **ld**(1)'s global symbol table. If any member of library **liba.a** defines this symbol, it is extracted (and perhaps other members are extracted as well). Without the **–u** option, there would have been no unresolved references or undefined symbols to cause **ld**(1) to search the archive library.

Dealing With Holes in Physical Memory

When memory configurations are defined in such a way that unconfigured areas exist in the virtual memory, each application or user must assume the responsibility of forming output sections that will fit into memory. For example, assume that memory is configured as follows:

```
MEMORY
{
        mem1:        o = 0x00000        l = 0x02000
        mem2:        o = 0x40000        l = 0x05000
        mem3:        o = 0x20000        l = 0x10000
}
```

Let the files **f1.o**, **f2.o**, . . . *fn*.**o** each contain three sections .**text**, .**data**, and .**bss**, and suppose the combined .**text** section is 0x12000 bytes. There is no configured area of memory in which this section can be placed. Appropriate directives must be supplied to break up the .**text** output section so **ld**(1) may do allocation. For example,

```
SECTIONS
{
        txt1:
        {
                f1.o ( .text)
                f2.o ( .text)
                f3.o ( .text)
        }
        txt2:
        {
                f4.o ( .text)
                f5.o ( .text)
                f6.o ( .text)
        }
        etc.
}
```

Allocation Algorithm

An output section is formed either as a result of a SECTIONS directive, by combining input sections of the same name, or by combining **.text** and **.init** into **.text**. An output section can have zero or more input sections comprising it. After the composition of an output section is determined, it must then be allocated into configured virtual memory. **ld**(1) uses an algorithm that attempts to minimize fragmentation of memory, and hence increases the possibility that a link edit run will be able to allocate all output sections within the specified virtual memory configuration. The algorithm proceeds as follows:

1. Any output sections for which explicit bonding addresses were specified are allocated.

2. Any output sections to be included in a specific named memory are allocated. In both this and the succeeding step, each output section is placed into the first available space within the (named) memory with any alignment taken into consideration.

3. Output sections not handled by one of the above steps are allocated.

If all memory is contiguous and configured (the default case), and no SECTIONS directives are given, then output sections are allocated in the order they appear to **ld**(1). Otherwise, output sections are allocated in the order they were defined or made known to **ld**(1) and put into the first available space they will fit in.

Incremental Link Editing

As previously mentioned, the output of **ld**(1) can be used as an input file to subsequent **ld**(1) runs providing that the relocation information is retained (**-r** option). Large applications may find it desirable to partition their C programs into subsystems, link each subsystem independently, and then link edit the entire application. For example,

Step 1:

ld –r –o outfile1 ifile1 infile1.o

```
    /* ifile1  */
    SECTIONS
    {
          ss1:
          {
                  f1.o
                  f2.o
                  . . .
                  fn.o
          }
    }
```

Step 2:

ld –r –o outfile2 ifile2 infile2.o

```
    /* ifile2  */
    SECTIONS
    {
          ss2:
          {
                  g1.o
                  g2.o
                  . . .
                  gn.o
          }
    }
```

Step 3:

>   **ld −a −o final.out outfile1 outfile2**

By judiciously forming subsystems, applications may achieve a form of incremental link editing whereby it is necessary to relink only a portion of the total link edit when a few files are recompiled.

To apply this technique, there are two simple rules:

■ Intermediate link edits should contain only SECTIONS declarations and be concerned only with the formation of output sections from input files and input sections. No binding of output sections should be done in these runs.

■ All allocation and memory directives, as well as any assignment statements, are included only in the final **ld**(1) call.

DSECT, COPY, NOLOAD, iNFO, and OVERLAY Sections

Sections may be given a type in a section definition as shown in the following example:

```
SECTIONS
{
        name1 0x200000 (DSECT)      : { file1.o }
        name2 0x400000 (COPY)       : { file2.o }
        name3 0x600000 (NOLOAD)     : { file3.o }
        name4          (INFO)       : { file4.o }
        name5 0x900000 (OVERLAY)    : { file5.o }


}
```

The DSECT option creates what is called a dummy section. A dummy section has the following properties:

- It does not participate in the memory allocation for output sections. As a result, it takes up no memory and does not show up in the memory map generated by **ld**(1).

- It may overlay other output sections and even unconfigured memory. DSECTs may overlay other DSECTs.

- The global symbols defined within the dummy section are relocated normally. That is, they appear in the output file's symbol table with the same value they would have had if the DSECT were actually loaded at its virtual address. DSECT-defined symbols may be referenced by other input sections. Undefined external symbols found within a DSECT cause specified archive libraries to be searched and any members which define such symbols are link edited normally (i.e., not as a DSECT).

- None of the section contents, relocation information, or line number information associated with the section is written to the output file.

In the above example, none of the sections from **file1.o** are allocated, but all symbols are relocated as though the sections were link edited at the specified address. Other sections could refer to any of the global symbols and they are resolved correctly.

A copy section created by the COPY option is similar to a dummy section. The only difference between a copy section and a dummy section is that the contents of a copy section and all associated information is written to the output file.

An INFO section is the same as a COPY section but its purpose is to carry information about the object file, whereas the COPY section may contain valid text and data. INFO sections are usually used to contain file version identification information.

A section with the type of NOLOAD differs in only one respect from a normal output section: its text and/or data is not written to the output file. A NOLOAD section is allocated virtual space, appears in the memory map, etc.

An OVERLAY section is relocated and written to the output file. It is different from a normal section in that it is not allocated and may overlay other sections or unconfigured memory.

Output File Blocking

The BLOCK option (applied to any output section or GROUP directive) is used to direct **ld**(1) to align a section at a specified byte offset in the output file. It has no effect on the address at which the section is allocated nor on any part of the link edit process. It is used purely to adjust the physical position of the section in the output file.

```
        SECTIONS
        {
                .text BLOCK(0x200) : { }
                .data ALIGN(0x20000) BLOCK(0x200) : { }
        }
```

With this SECTIONS directive, **ld**(1) assures that each section, **.text** and **.data**, is physically written at a file offset, which is a multiple of 0x200 (e.g., at an offset of 0, 0x200, 0x400, and so forth, in the file).

Nonrelocatable Input Files

If a file produced by **ld**(1) is intended to be used in a subsequent **ld**(1) run, the first **ld**(1) run should have the −r option set. This preserves relocation information and permits the sections of the file to be relocated by the subsequent run.

If an input file to **ld**(1) does not have relocation or symbol table information [perhaps from the action of a **strip**(1) command, or from being link edited without a −r option, or with a −s option], the link edit run continues using the nonrelocatable input file.

For such a link edit to be successful (i.e., to actually and correctly link edit all input files, relocate all symbols, resolve unresolved references, etc.), two conditions on the nonrelocatable input files must be met.

■ Each input file must have no unresolved external references.

■ Each input file must be bound to the exact same virtual address as it was bound to in the **ld**(1) run that created it.

| NOTE | If these two conditions are not met for all nonrelocatable input files, no error messages are issued. Because of this fact, extreme care must be taken when supplying such input files to **ld**(1). |

Syntax Diagram for Input Directives

Directives	Expanded Directives
<ifile>	{ <cmd> }
<cmd>	<memory>
	<sections>
	<assignment>
	<filename>
	<flags>
<memory>	MEMORY { <memory_spec>
	{ [,] <memory_spec> } }
<memory_spec>	<name> [<attributes>] :
	<origin_spec> [,] <length_spec>
<attributes>	({ R ┃ W ┃ X ┃ I })
<origin_spec>	<origin> = <long>
<length_spec>	<length> = <long>
<origin>	ORIGIN ┃ o ┃ org ┃ origin
<length>	LENGTH ┃ l ┃ len ┃ length

Figure 12-2: Syntax Diagram for Input Directives (Sheet 1 of 4)

NOTE Two punctuation symbols, brackets and braces, do double duty in this diagram.

Where the actual symbols, [] and {} are used, they are part of the syntax and must be present when the directive is specified.

Where you see the symbols [and] (larger and in bold), it means the material enclosed is optional.

Where you see the symbols { and } (larger and in bold), it means multiple occurrences of the material enclosed are permitted.

Directives	Expanded Directives
<sections>	SECTIONS { {<sec_or_group>} }
<sec_or_group>	<section> ¦ <group> ¦ <library>
<group>	GROUP <group_options> : {
	<section_list> } [<mem_spec>]
<section_list>	<section> { [,] <section> }
<section>	<name> <sec_options> :
	{ <statement> }
	[<fill>] [<mem_spec>]
<group_options>	[<addr>] ¦ [<align_option>] [<block_option>]
<sec_options>	[<addr>] ¦ [<align_option>]
	[<block_option>] [<type_option>]
<addr>	<long> ¦ <bind>(<expr>)
<alignoption>	<align> (<expr>)
<align>	ALIGN ¦ align
<block_option>	<block> (<long>)
<block>	BLOCK ¦ block
<type_option>	(DSECT) ¦ (NOLOAD) ¦ (COPY)
	¦ (INFO) ¦ (OVERLAY)
<fill>	= <long>
<mem_spec>	> <name>
	> <attributes>
<statement>	<filename>
	<filename> (<name_list>) ¦ [COMMON]
	* (<name_list>) ¦ [COMMON]
	<assignment>
	<library>
	null

Figure 12-2: Syntax Diagram for Input Directives (Sheet 2 of 4)

Directives	Expanded Directives
<name_list>	<section_name> **[,]** { <section_name> }
<library>	–l<name>
<bind>	BIND ┆ bind
<assignment>	<lside> <assign_op> <expr> <end>
<lside>	<name> ┆ .
<assign_op>	= ┆ += ┆ –= ┆ *= ┆/ =
<end>	; ┆ ,
<expr>	<expr> <binary_op> <expr>
	<term>
<binary_op>	* ┆ / ┆ %
	+ ┆ –
	>> ┆ <<
	== ┆ != ┆ > ┆ < ┆ <= ┆ >=
	&
	┆
	&&
	┆┆
<term>	<long>
	<name>
	<align> (<term>)
	(<expr>)
	<unary_op> <term>
	<phy> (<lside>)
	<sizeof>(<sectionname>)
	<next>(<long>)
	<addr>(<sectionname>)
<unary_op>	! ┆ –
<phy>	PHY ┆ phy
<sizeof>	SIZEOF ┆ sizeof

Figure 12-2: Syntax Diagram for Input Directives (Sheet 3 of 4)

Directives	Expanded Directives
<next>	NEXT ┆ next
<addr>	ADDR ┆ addr
<flags>	–e<wht_space><name>
	–f<wht_space><long>
	–h<wht_space><long>
	–l<name>
	–m
	–o<wht_space><filename>
	–r
	–s
	–t
	–u<wht_space><name>
	–z
	–H
	–L<path_name>
	–M
	–N
	–S
	–V
	–VS<wht_space><long>
	–a
	–x
<name>	Any valid symbol name
<long>	Any valid long integer constant
<wht_space>	Blanks, tabs, and newlines
<filename>	Any valid UNIX Operating System file name. This may include a full or partial path name.
<sectionname>	Any valid section name, up to 8 characters
<path_name>	Any valid UNIX Operating System path name (full or partial)

Figure 12-2: Syntax Diagram for Input Directives (Sheet 4 of 4)

13 make

Introduction

The trend toward increased modularity of programs means that a project may have to cope with a large assortment of individual files. There may also be a wide range of generation procedures needed to turn the assortment of individual files into the final executable product.

make(1), documented in the *Programmer's Reference Manual*, provides a method for maintaining up-to-date versions of programs that consist of a number of files that may be generated in a variety of ways:

An individual programmer can easily forget

■ file-to-file dependencies

■ files that were modified and the impact that has on other files

■ the exact sequence of operations needed to generate a new version of the program

In a description file, **make** keeps track of the commands that create files and the relationship between files. Whenever a change is made in any of the files that make up a program, the **make** command creates the finished program by recompiling only those portions directly or indirectly affected by the change.

The basic operation of **make** is to

■ find the target in the description file

■ ensure that all the files on which the target depends, the files needed to generate the target, exist and are up-to-date

■ create the target file if any of the generators have been modified more recently than the target

The description file that holds the information on interfile dependencies and command sequences is conventionally called **makefile**, **Makefile**, or **s.[mM]akefile**. If this naming convention is followed, the simple command **make** is usually sufficient to regenerate the target regardless of the number of files edited since the last **make**. In most cases, the description file is not difficult to write and changes infrequently. Even if only a single file has been edited, rather than typing all the commands to regenerate the target, typing the **make** command ensures the regeneration is done in the prescribed way.

Basic Features

The basic operation of **make** is to update a target file by ensuring that all of the files on which the target file depends exist and are up-to-date. The target file is regenerated if it has not been modified since the dependents were modified. The **make** program searches the graph of dependencies. The operation of **make** depends on its ability to find the date and time that a file was last modified.

The **make** program operates using three sources of information:

- a user-supplied description file

- file names and last-modified times from the file system

- built-in rules to bridge some of the gaps

To illustrate, consider a simple example in which a program named **prog** is made by compiling and loading three C language files **x.c**, **y.c**, and **z.c** with the **math** library. By convention, the output of the C language compilations will be found in files named **x.o**, **y.o**, and **z.o**. Assume that the files **x.c** and **y.c** share some declarations in a file named **defs.h**, but that **z.c** does not. That is, **x.c** and **y.c** have the line

        #include "defs.h"

The following specification describes the relationships and operations:

        prog : x.o y.o z.o
                cc x.o y.o z.o   -lm -o prog

        x.o y.o :   defs.h

If this information were stored in a file named **makefile**, the command

   make

would perform the operations needed to regenerate **prog** after any changes had been made to any of the four source files **x.c**, **y.c**, **z.c**, or **defs.h**. In the example above, the first line states that **prog** depends on three **.o** files. Once these object files are current, the second line describes how to load them to create **prog**. The third line states that **x.o** and **y.o** depend on the file **defs.h**. From the file system, **make** discovers that there are three **.c** files corresponding to the needed **.o** files and uses built-in rules on how to generate an object from a C source file (i.e., issue a **cc -c** command).

If **make** did not have the ability to determine automatically what needs to be done, the following longer description file would be necessary:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  -lm  -o  prog
x.o :  x.c  defs.h
        cc  -c  x.c
y.o :  y.c  defs.h
        cc  -c  y.c
z.o :  z.c
        cc  -c  z.c
```

If none of the source or object files have changed since the last time **prog** was made, and all of the files are current, the command **make** announces this fact and stops. If, however, the **defs.h** file has been edited, **x.c** and **y.c** (but not **z.c**) are recompiled; and then **prog** is created from the new **x.o** and **y.o** files, and the existing **z.o** file. If only the file **y.c** had changed, only it is recompiled; but it is still necessary to reload **prog**. If no target name is given on the **make** command line, the first target mentioned in the description is created; otherwise, the specified targets are made. The command

> **make x.o**

would regenerate **x.o** if **x.c** or **defs.h** had changed.

A method often useful to programmers is to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros (for information about macros, see "Description Files and Substitutions" further along in this chapter.) Thus, an entry "save" might be included to copy a certain set of files, or an entry "clean" might be used to throw away unneeded intermediate files.

If a file exists after such commands are executed, the file's time of last modification is used in further decisions. If the file does not exist after the commands are executed, the current time is used in making further decisions.

You can maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

A simple macro mechanism for substitution in dependency lines and command strings is used by **make**. Macros can either be defined by command-line arguments or included in the description file. In either case, a macro consists of a name followed by an equals sign followed by what the macro stands for. A macro is invoked by preceding the name by a dollar sign. Macro names longer than one character must be parenthesized. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two are equivalent.

$*, $@, $?, and $< are four special macros that change values during the execution of the command. (These four macros are described later in this chapter under "Description Files and Substitutions.") The following fragment shows assignment and use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = —lm
prog: $(OBJECTS)
        cc $(OBJECTS)   $(LIBES)   —o prog
    . . .
```

The command

    make LIBES="–ll –lm"

loads the three objects with both the **lex** (–ll) and the **math** (–lm) libraries, because macro definitions on the command line override definitions in the description file. (In UNIX System commands, arguments with embedded blanks must be quoted.)

As an example of the use of **make**, a description file that might be used to maintain the **make** command itself is given. The code for **make** is spread over a number of C language source files and has a **yacc** grammar. An example of the description file follows:

```
# Description file for the make command

FILES = Makefile defs.h main.c daname.c misc.c
        files.c dosys.c gram.y
OBJECTS = main.o daname.o misc.o files.o
          dosys.o gram.o
LIBES= -lld
LINT = lint -p
CFLAGS = -O
LP = /usr/bin/lp

make:   $(OBJECTS)
        $(CC) $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        @size make

$(OBJECTS):   defs.h

cleanup:
        -rm *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make && rm make

lint :  dosys.c daname.c files.c main.c misc.c gram.c
        $(LINT) dosys.c daname.c files.c main.c misc.c \
        gram.c

                # print files that are out-of-date
                # with respect to "print" file.

print:  $(FILES)
        pr $? | $(LP)
        touch print
```

The **make** program prints out each command before issuing it.

The following output results from typing the command **make** in a directory containing only the source and description files:

```
cc  -o -c main.c
cc  -o -c daname.c
cc  -o -c misc.c
cc  -o -c files.c
cc  -o -c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  -o -c gram.c
cc  main.o daname.o misc.o files.o dosys.o
    gram.o  -lld -o make
13188 + 3348 + 3044 = 19580
```

The string of digits results from the **size make** command. The printing of the command line itself was suppressed by an at sign, **@**, in the description file.

Description Files and Substitutions

The following section will explain the customary elements of the description file.

Comments

The comment convention is that a sharp, #, and all characters on the same line after a sharp are ignored. Blank lines and lines beginning with a sharp are totally ignored.

Continuation Lines

If a noncomment line is too long, the line can be continued by using a backslash. If the last character of a line is a backslash, then the backslash, the new line, and all following blanks and tabs are replaced by a single blank.

Macro Definitions

A macro definition is an identifier followed by an equal sign. The identifier must not be preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped). The following are valid macro definitions:

```
2 = xyz
abc = -11 -ly -lm
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as its value. Remember, however, that some macros are explicitly defined in **make**'s own rules. (See Figure 13-2 at the end of the chapter.)

General Form

The general form of an entry in a description file is

```
target1 [target2 ...] :[:] [dependent1 ...] [; commands] [# ...]
[ \t commands] [# ...]
    . . .
```

Items inside brackets may be omitted and targets and dependents are strings of letters, digits, periods, and slashes. Shell metacharacters such as * and ? are expanded when the line is evaluated. Commands may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a dependency line. A command is any string of characters not including a sharp, #, except when the sharp is in quotes.

Dependency Information

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type. For the more common single-colon case, a command sequence may be associated with at most one dependency line. If the target is out-of-date with any of the dependents on any of the lines and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise, a default rule may be invoked. In the double-colon case, a command sequence may be associated with more than one dependency line. If the target is out-of-date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. The double colon form is particularly useful in updating archive-type files, where the target is the archive library itself. (An example is included in the "Archive Libraries" section later in this chapter.)

Executable Commands

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the silent mode (-s option of the **make** command) or if the command line in the description file begins with an @ sign. **make** normally stops if any command

signals an error by returning a nonzero error code. Errors are ignored if the **–i** flag has been specified on the **make** command line, if the fake target name .IGNORE appears in the description file, or if the command string in the description file begins with a hyphen. If a program is known to return a meaningless status, a hyphen in front of the command that invokes it is appropriate. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., **cd** and shell control commands) that have meaning only within a single shell process. These results are forgotten before the next line is executed.

Before issuing any command, certain internally maintained macros are set. The **$@** macro is set to the full target name of the current target. The **$@** macro is evaluated only for explicitly named dependencies. The **$?** macro is set to the string of names that were found to be younger than the target. The **$?** macro is evaluated when explicit rules from the **makefile** are evaluated. If the command was generated by an implicit rule, the **$<** macro is the name of the related file that caused the action; and the **$*** macro is the prefix shared by the current and the dependent file names. If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name DEFAULT are used. If there is no such name, **make** prints a message and stops.

In addition, a description file may also use the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)** (see below).

Extensions of $*, $@, and $<

The internally generated macros **$***, **$@**, and **$<** are useful generic terms for current targets and out-of-date relatives. To this list has been added the following related macros: **$(@D)**, **$(@F)**, **$(*D)**, **$(*F)**, **$(<D)**, and **$(<F)**. The **D** refers to the directory part of the single character macro. The **F** refers to the file name part of the single character macro. These additions are useful when building hierarchical **makefiles**. They allow access to directory names for purposes of using the **cd** command of the shell. Thus, a command can be

    cd $(<D); $(MAKE) $(<F)

Output Translations

Macros in shell commands are translated when evaluated. The form is as follows:

```
$(macro:string1=string2)
```

The meaning of **$(macro)** is evaluated. For each appearance of **string1** in the evaluated macro, **string2** is substituted. The meaning of finding **string1** in **$(macro)** is that the evaluated **$(macro)** is considered as a series of strings, each delimited by white space (blanks or tabs). Thus, the occurrence of **string1** in **$(macro)** means that a regular expression of the following form has been found:

```
.*<string1>[TAB|BLANK]
```

This particular form was chosen because **make** usually concerns itself with suffixes. The usefulness of this type of translation occurs when maintaining archive libraries. Now, all that is necessary is to accumulate the out-of-date members and write a shell script which can handle all the C language programs (i.e., those files ending in **.c**). Thus, the following fragment optimizes the executions of **make** for maintaining an archive library:

```
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)(c.o)
        $(CC) -c $(CFLAGS) $(?:.o=.c)
        $(AR) $(ARFLAGS) $(LIB) $?
        rm $?
```

A dependency of the preceding form is necessary for each of the different types of source files (suffixes) that define the archive library. These translations are added in an effort to make more general use of the wealth of information that **make** generates.

Recursive Makefiles

Another feature of **make** concerns the environment and recursive invocations. If the sequence $(MAKE) appears anywhere in a shell command line, the line is executed even if the **-n** flag is set. Since the **-n** flag is exported across invocations of **make** (through the MAKEFLAGS variable), the only thing that is executed is the **make** command itself. This feature is useful when a hierarchy of **makefile**(s) describes a set of software subsystems. For testing purposes, **make -n ...** can be executed and everything that would have been done will be printed, including output from lower level invocations of **make**.

Suffixes and Transformation Rules

make uses an internal table of rules to learn how to transform a file with one suffix into a file with another suffix. If the **-r** flag is used on the **make** command line, the internal table is not used.

The list of suffixes is actually the dependency list for the name .SUFFIXES. **make** searches for a file with any of the suffixes on the list. If it finds one, **make** transforms it into a file with another suffix. The transformation rule names are the concatenation of the before and after suffixes. The name of the rule to transform a **.r** file to a **.o** file is thus **.r.o**. If the rule is present and no explicit command sequence has been given in the user's description files, the command sequence for the rule **.r.o** is used. If a command is generated by using one of these suffixing rules, the macro $* is given the value of the stem (everything but the suffix) of the name of the file to be made; and the macro $< is the full name of the dependent that caused the action.

The order of the suffix list is significant since the list is scanned from left to right. The first name formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can add an entry for .SUFFIXES in the description file. The dependents are added to the usual list. A .SUFFIXES line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed.

Implicit Rules

make uses a table of suffixes and a set of transformation rules to supply default dependency information and implied commands. The default suffix list is as follows:

.o   object file

.c   C source file

.c~   SCCS C source file

.f   FORTRAN source file

.f~   SCCS FORTRAN source file

.s   assembler source file

.s~   SCCS Assembler source file

.y   **yacc** source grammar

.y~   SCCS **yacc** source grammar

.l   **lex** source grammar

.l~   SCCS **ex** source grammar

.h   header file

.h~   SCCS header file

.sh   shell file

.sh~   SCCS shell file

Figure 13-1 summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.
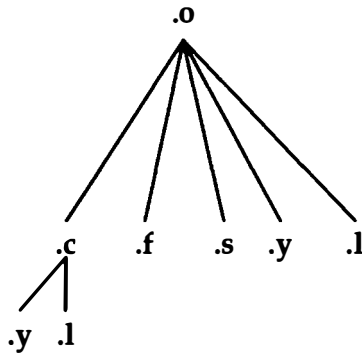
Figure 13-1: Summary of Default Transformation Path

If the file **x.o** is needed and an **x.c** is found in the description or directory, the **x.o** file would be compiled. If there is also an **x.l**, that source file would be run through **lex** before compiling the result. However, if there is no **x.c** but there is an **x.l**, **make** would discard the intermediate C language file and use the direct link as shown in Figure 13-1.

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, F77, YACC, and LEX. The command

> **make CC=newcc**

will cause the **newcc** command to be used instead of the usual C language compiler. The macros ASFLAGS, CFLAGS, F77FLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus

> **make "CFLAGS=-g"**

causes the **cc** command to include debugging information.

Archive Libraries

The **make** program has an interface to archive libraries. A user may name a member of a library in the following manner:

projlib(object.o)

or

projlib((entrypt))

The second method actually refers to an entry point of an object file within the library. (**make** looks through the library, locates the entry point, and translates it to the correct object file name.)

To use this procedure to maintain an archive library, the following type of **makefile** is required:

```
projlib::   projlib(pfile1.o)
            $(CC) -c -O pfile1.c
            $(AR) $(ARFLAGS) projlib pfile1.o
            rm pfile1.o
projlib::   projlib(pfile2.o)
            $(CC) -c -O pfile2.c
            $(AR) $(ARFLAGS) projlib pfile2.o
            rm pfile2.o
```

and so on for each object.

This is tedious and error-prone. Obviously, the command sequences for adding a C language file to a library are the same for each invocation; the file name being the only difference each time. (This is true in most cases.)

The **make** command also gives the user access to a rule for building libraries. The handle for the rule is the **.a** suffix. Thus, a **.c.a** rule is the rule for compiling a C language source file, adding it to the library, and removing the **.o** cadaver. Similarly, the **.y.a**, the **.s.a**, and the **.l.a** rules rebuild **yacc**, assembler, and **lex** files, respectively. The archive rules defined internally are **.c.a**, **.c~.a**, **.f.a**, **.f~.a**, and **.s~.a**. (The tilde, ~, syntax will be described shortly.) The user may define other needed rules in the description file.

The two-member library mentioned earlier is then maintained with the following shorter **makefile**:

```
projlib:        projlib(pfile1.o) projlib(pfile2.o)
                @echo projlib up-to-date
```

The internal rules are already defined to complete the preceding library maintenance. The actual **.c.a** rule is as follows:

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o
```

Thus, the **$@** macro is the **.a** target (**projlib**); the **$<** and **$*** macros are set to the out-of-date C language file; and the file name minus the suffix, respectively (**pfile1.c** and **pfile1**). The **$<** macro (in the preceding rule) could have been changed to **$*.c**.

It might be useful to go into some detail about exactly what **make** does when it sees the construction

```
projlib:    projlib(pfile1.o)
            @echo projlib up-to-date
```

Assume the object in the library is out-of-date with respect to **pfile1.c**. Also, there is no **pfile1.o** file.

1. **make projlib**.

2. Before **make**ing **projlib**, check each dependent of **projlib**.

3. **projlib(pfile1.o)** is a dependent of **projlib** and needs to be generated.

4. Before generating **projlib(pfile1.o)**, check each dependent of **projlib(pfile1.o)**. (There are none.)

5. Use internal rules to try to create **projlib(pfile1.o)**. (There is no explicit rule.) Note that **projlib(pfile1.o)** has a parenthesis in the name to identify the target suffix as **.a**. This is the key. There is no explicit **.a** at the end of the **projlib** library name. The parenthesis implies the **.a** suffix. In this sense, the **.a** is hard-wired into **make**.

6. Break the name **projlib(pfile1.o)** up into **projlib** and **pfile1.o**. Define two macros, **$@** (=**projlib**) and **$*** (=**pfile1**).

7. Look for a rule .*X*.a and a file $*.*X*. The first .*X* (in the .SUFFIXES list) which fulfills these conditions is **.c** so the rule is **.c.a**, and the file is **pfile1.c**. Set $< to be **pfile1.c** and execute the rule. In fact, **make** must then compile **pfile1.c**.

8. The library has been updated. Execute the command associated with the **projlib:** dependency; namely

        @echo projlib up-to-date

It should be noted that to let **pfile1.o** have dependencies, the following syntax is required:

        projlib(pfile1.o): $(INCDIR)/stdio.h   pfile1.c

There is also a macro for referencing the archive member name when this form is used. The $% macro is evaluated each time $@ is evaluated. If there is no current archive member, $% is null. If an archive member exists, then $% evaluates to the expression between the parenthesis.

Source Code Control System File Names: the Tilde

The syntax of **make** does not directly permit referencing of prefixes. For most types of files on UNIX Operating System machines, this is acceptable since nearly everyone uses a suffix to distinguish different types of files. The SCCS files are the exception. Here, **s.** precedes the file name part of the complete path name.

To allow **make** easy access to the prefix **s.** the tilde, ~, is used as an identifier of SCCS files. Hence, **.c~.o** refers to the rule which transforms an SCCS C language source file into an object file. Specifically, the internal rule is

```
.c~.o:
        $(GET) $(GFLAGS) $<
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
```

Thus, the tilde appended to any suffix transforms the file search into an SCCS file name search with the actual suffix named by the dot and all characters up to (but not including) the tilde.

The following SCCS suffixes are internally defined:

```
.c~
.f~
.y~
.l~
.s~
.sh~
.h~
```

The following rules involving SCCS transformations are internally defined:

```
           .c~:
           .f~:
           .sh~:
           .c~.a:
           .c~.c:
           .c~.o:
           .f~.a:
           .f~.f:
           .f~.o:
           .s~.a:
           .s~.s:
           .s~.o:
           .y~.c:
           .y~.o:
           .l~.l:
           .l~.o:
           .h~.h:
```

Obviously, the user can define other rules and suffixes, which may prove use-
ful. The tilde provides a handle on the SCCS file name format so that this is
possible.

The Null Suffix

There are many programs that consist of a single source file. **make** han-
dles this case by the null suffix rule. Thus, to maintain the UNIX System pro-
gram **cat**, a rule in the **makefile** of the following form is needed:

```
    .c:
             $(CC) $(CFLAGS) $< −o $@
```

In fact, this **.c:** rule is internally defined, so no **makefile** is necessary at all.
The user only needs to type

      make cat dd echo date

(these are all UNIX System single-file programs) and all four C language
source files are passed through the above shell command line associated with
the **.c:** rule. The internally defined single suffix rules are

```
.c:
.c~:
.f:
.f~:
.sh:
.sh~:
```

Others may be added in the **makefile** by the user.

include **Files**

The **make** program has a capability similar to the **#include** directive of the C preprocessor. If the string **include** appears as the first seven letters of a line in a **makefile** and is followed by a blank or a tab, the rest of the line is assumed to be a file name, which the current invocation of **make** will read. Macros may be used in file names. The file descriptors are stacked for reading **include** files so that no more than 16 levels of nested **include**s are supported.

SCCS Makefiles

Makefiles under SCCS control are accessible to **make**. That is, if **make** is typed and only a file named **s.makefile** or **s.Makefile** exists, **make** will do a **get** on the file, then read and remove the file.

Dynamic Dependency Parameters

The parameter has meaning only on the dependency line in a makefile. The **$$@** refers to the current "thing" to the left of the colon (which is **$@**). Also the form **$$(@F)** exists, which allows access to the file part of **$@**. Thus, in the following:

```
cat:      $$@.c
```

the dependency is translated at execution time to the string **cat.c**. This is useful for building a large number of executable files, each of which has only one source file. For instance, the UNIX System software command directory could have a **makefile** like:

```
CMDS = cat dd echo date cmp comm chown

$(CMDS):          $$@.c
        $(CC) -o $? -o $@
```

Obviously, this is a subset of all the single file programs. For multiple file programs, a directory is usually allocated and a separate **makefile** is made. For any particular file that has a peculiar compilation procedure, a specific entry must be made in the **makefile**.

The second useful form of the dependency parameter is **$$(@F)**. It represents the file name part of **$$@**. Again, it is evaluated at execution time. Its usefulness becomes evident when trying to maintain the **/usr/include** directory from a makefile in the **/usr/src/head** directory. Thus, the **/usr/src/head/makefile** would look like this.

```
INCDIR = /usr/include

INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCDIR)/dir.h \
        $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
        cp $? $@
        chmod 0444 $@
```

This would completely maintain the **/usr/include** directory whenever one of the above files in **/usr/src/head** was updated.

Command Usage

The **make** command description is found under **make(1)** in the *Programmer's Reference Manual*.

The make Command

The **make** command takes macro definitions, options, description file names, and target file names as arguments in the form:

make [*options*] [*macro definitions*] [*targets*]

The following summary of command operations explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files. Next, the option arguments are examined. The permissible options are as follows:

-i   ignores error codes returned by invoked commands. This mode is entered if the fake target name .IGNORE appears in the description file.

-s   is silent mode. Does not print command lines before executing. This mode is also entered if the fake target name .SILENT appears in the description file.

-r   Does not use the built-in rules.

-n   is no execute mode. Prints commands, but does not execute them. Even lines beginning with an **@** sign are printed.

-t   touches the target files (causing them to be up-to-date) rather than issue the usual commands.

-q   is a question. The **make** command returns a zero or nonzero status code depending on whether the target file is or is not up-to-date.

-p   prints out the complete set of macro definitions and target descriptions.

-k   abandons work on the current entry if something goes wrong, but continues on other branches that do not depend on the current entry.

-e   are environment variables that override assignments within **makefiles**.

-f   is a description file name. The next argument is assumed to be the name of a description file. A file name of – denotes the standard input. If there are no **-f** arguments, the file named **makefile** or **Makefile** or **s.[mM]akefile** in the current directory is read. The contents of the description files override the built-in rules if they are present.

The following two arguments are evaluated in the same manner as flags:

.DEFAULT   If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name .DEFAULT are used if it exists.

.PRECIOUS   Dependents on this target are not removed when quit or interrupt is pressed.

Finally, the remaining arguments are assumed to be the names of targets to be made and the arguments are done in left-to-right order. If there are no such arguments, the first name in the description file that does not begin with a period is made.

Environment Variables

Environment variables are read and added to the macro definitions each time **make** executes. Precedence is a prime consideration in doing this properly. The following describes **make**'s interaction with the environment. A macro, MAKEFLAGS, is maintained by **make**. The macro is defined as the collection of all input flag arguments into a string (without minus signs). The macro is exported and thus accessible to further invocations of **make**. Command line flags and assignments in the **makefile** update MAKEFLAGS. Thus, to describe how the environment interacts with **make**, the MAKEFLAGS macro (environment variable) must be considered.

When executed, **make** assigns macro definitions in the following order:

1. Read the MAKEFLAGS environment variable. If it is not present or null, the internal **make** variable MAKEFLAGS is set to the null string. Otherwise, each letter in MAKEFLAGS is assumed to be an input flag argument and is processed as such. (The only exceptions are the **-f**, **-p**, and **-r** flags.)

2. Read the internal list of macro definitions.

3. Read the environment. The environment variables are treated as macro definitions and marked as **exported** (in the shell sense).

4. Read the **makefile**(s). The assignments in the **makefile**(s) overrides the environment. This order is chosen so that when a **makefile** is read and executed, you know what to expect. That is, you get what is seen unless the −e flag is used. The −e is the line flag, which tells **make** to have the environment override the **makefile** assignments. Thus, if **make −e …** is typed, the variables in the environment override the definitions in the **makefile**. Also MAKEFLAGS override the environment if assigned. This is useful for further invocations of **make** from the current **makefile**.

It may be clearer to list the precedence of assignments. Thus, in order from least binding to most binding, the precedence of assignments is as follows:

1. internal definitions

2. environment

3. **makefile**(s)

4. command line

The −e flag has the effect of rearranging the order to

1. internal definitions

2. **makefile**(s)

3. environment

4. command line

This order is general enough to allow a programmer to define a **makefile** or set of **makefiles** whose parameters are dynamically definable.

Suggestions and Warnings

The most common difficulties arise from **make**'s specific meaning of dependency. If file **x.c** has a

        #include "defs.h"

line, then the object file **x.o** depends on **defs.h**; the source file **x.c** does not. If **defs.h** is changed, nothing is done to the file **x.c** while file **x.o** must be recreated.

To discover what **make** would do, the –n option is very useful. The command

   make –n

orders **make** to print out the commands that **make** would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be mild in character (e.g., adding a comment to an **include** file), the –t (touch) option can save a lot of time. Instead of issuing a large number of superfluous recompilations, **make** updates the modification times on the affected file. Thus, the command

   make –ts

(touch silently) causes the relevant files to appear up-to-date. Obvious care is necessary because this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

Internal Rules

The standard set of internal rules used by **make** are reproduced below.

```
#
#               SUFFIXES RECOGNIZED BY MAKE
#
.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .h .h~ .sh .sh~ .f .f~

#
#               PREDEFINED MACROS
#
MAKE=make
AR=ar
ARFLAGS=-rv
AS=as
ASFLAGS=
CC=cc
CFLAGS=-O
F77=f77
F77FLAGS=
GET=get
GFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
YACC=yacc
YFLAGS=
```

Figure 13-2: **make** Internal Rules (Sheet 1 of 5)

```
#
#              SINGLE SUFFIX RULES
#
.c:
               $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@
.c~ :
               $(GET) $(GFLAGS) $<
               $(CC) $(CFLAGS) $(LDFLAGS) $*.c -o $*
               -rm -f $*.c
.f:
               $(F77) $(F77FLAGS) $(LDFLAGS) $< -o $@
.f~ :
               $(GET) $(GFLAGS) $<
               $(F77) $(F77FLAGS) $(LDFLAGS) $< -o $*
               -rm -f $*.f
.sh:
               cp $< $@; chmod 0777 $@
.sh~ :
               $(GET) $(GFLAGS) $<
               cp $*.sh $*; chmod 0777 $@
               -rm -f $*.sh
```

Figure 13-2: **make** Internal Rules (Sheet 2 of 5)

```
#
#               DOUBLE SUFFIX RULES
#
.c~.c  .f~.f  .s~.s  .sh~.sh  .y~.y  .l~.l  .h~.h:
                $(GET) $(GFLAGS) $<

.c.a:
                $(CC) -c $(CFLAGS) $<
                $(AR) $(ARFLAGS) $@ $*.o
                rm -f $*.o
.c~.a:
                $(GET) $(GFLAGS) $<
                $(CC) -c $(CFLAGS) $*.c
                $(AR) $(ARFLAGS) $@ $*.o
                rm -f $*.[co]

.c.o:
                $(CC) $(CFLAGS) -c $<
.c~.o:
                $(GET) $(GFLAGS) $<
                $(CC) $(CFLAGS) -c $*.c
                -rm -f $*.c

.f.a:
                $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
                $(AR) $(ARFLAGS) $@ $*.o
                -rm -f $*.o
.f~.a;
                $(GET) $(GFLAGS) $<
                $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
                $(AR) $(ARFLAGS) $@ $*.o
                -rm -f $*.[fo]
```

Figure 13-2: **make** Internal Rules (Sheet 3 of 5)

```
.f.o:
                $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
.f~.o:

                $(GET) $(GFLAGS) $<
                $(F77) $(F77FLAGS) $(LDFLAGS) -c $*.f
                -rm -f $*.f
.s~.a:
                $(GET) $(GFLAGS) $<
                $(AS) $(ASFLAGS) -o $*.o $*.s
                $(AR) $(ARFLAGS) $@ $*.o
                -rm -f $*.[so]
.s.o:
                $(AS) $(ASFLAGS) -o $@ $<
.s~.o:

                $(GET) $(GFLAGS) $<
                $(AS) $(ASFLAGS) -o $*.o $*.s
                -rm -f $*.s
.l.c :
                $(LEX) $(LFLAGS) $<
                mv lex.yy.c $@
.l~.c:

                $(GET) $(GFLAGS) $<
                $(LEX) $(LFLAGS) $*.l
                mv lex.yy.c $@
```

Figure 13-2: **make** Internal Rules (Sheet 4 of 5)

```
.l.o:
                $(LEX) $(LFLAGS) $<
                $(CC) $(CFLAGS) -c lex.yy.c
                rm lex.yy.c
                mv lex.yy.o $@
                -rm -f $*.l
.l~.o:
                $(GET) $(GFLAGS) $<
                $(LEX) $(LFLAGS) $*.l
                $(CC) $(CFLAGS) -c lex.yy.c
                rm -f lex.yy.c $*.l
                mv lex.yy.o $*.o
.y.c :
                $(YACC) $(YFLAGS) $<
                mv y.tab.c $@
.y~.c :
                $(GET) $(GFLAGS) $<
                $(YACC) $(YFLAGS) $*.y
                mv y.tab.c $*.c
                -rm -f $*.y
.y.o:
                $(YACC) $(YFLAGS) $<
                $(CC) $(CFLAGS) -c y.tab.c
                rm y.tab.c
                mv y.tab.o $@
.y~.o:
                $(GET) $(GFLAGS) $<
                $(YACC) $(YFLAGS) $*.y
                $(CC) $(CFLAGS) -c y.tab.c
                rm -f y.tab.c $*.y
                mv y.tab.o $*.o
```

Figure 13-2: **make** Internal Rules (Sheet 5 of 5)

14 Source Code Control System (SCCS)

SCCS Files

Introduction

The Source Code Control System (SCCS) is a maintenance and enhancement tracking tool that runs under the UNIX System. SCCS takes custody of a file and, when changes are made, identifies and stores them in the file with the original source code and/or documentation. As other changes are made, they too are identified and retained in the file.

Retrieval of the original or any set of changes is possible. Any version of the file as it develops can be reconstructed for inspection or additional modification. History data can be stored with each version: why the changes were made, who made them, and when they were made.

This guide covers the following:

■ SCCS for Beginners: how to make, retrieve, and update an SCCS file

■ Delta Numbering: how versions of an SCCS file are named

■ SCCS Command Conventions: what rules apply to SCCS commands

■ SCCS Commands: the fourteen SCCS commands and their more useful arguments

■ SCCS Files: protection, format, and auditing of SCCS files

Neither the implementation of SCCS nor the installation procedure for SCCS is described in this guide.

SCCS For Beginners

Several terminal session fragments are presented in this section. Try them all. The best way to learn SCCS is to use it.

Terminology

A delta is a set of changes made to a file under SCCS custody. To identify and keep track of a delta, it is assigned an SID (SCCS IDentification) number. The SID for any original file turned over to SCCS is composed of release number 1 and level number 1, stated as 1.1. The SID for the first set of changes made to that file, that is, its first delta, is release 1 version 2, or 1.2. The next delta would be 1.3, the next 1.4, and so on. More on delta numbering later. At this point, it is enough to know that by default SCCS assigns SIDs automatically.

Creating an SCCS File by means of admin

Suppose, for example, you have a file called **lang** that is simply a list of five programming language names. Use a text editor to create file **lang** containing the following list.

```
C
PL/1
FORTRAN
COBOL
ALGOL
```

Custody of your **lang** file can be given to SCCS using the **admin** command (i.e., administer SCCS file). The following creates an SCCS file from the **lang** file:

> **admin –ilang s.lang**

All SCCS files must have names that begin with **s.**, hence **s.lang**. The **–i** key letter, together with its value **lang**, means **admin** is to create an SCCS file and initialize it with the contents of the file **lang**.

The **admin** command replies

```
No id keywords (cm7)
```

This is a warning message that may also be issued by other SCCS commands. Ignore it for now. Its significance is described later with the **get** command under "SCCS Commands." In the following examples, this warning message is not shown although it may be issued.

Remove the **lang** file. It is no longer needed because it exists now under SCCS as **s.lang**.

rm lang

Retrieving a File by means of get

Use the **get** command as follows:

get s.lang

This retrieves **s.lang** and prints

```
1.1
5 lines
```

This tells you that **get** retrieved version 1.1 of the file, which is made up of five lines of text.

The retrieved text has been placed in a new file known as a "g.file." SCCS forms the g.file name by deleting the prefix **s.** from the name of the SCCS file. Thus, the original **lang** file has been recreated.

If you list, **ls**(1) (documented in the *User's/System Administrator's Reference Manual*), the contents of your directory, you will see both **lang** and **s.lang**. SCCS retains **s.lang** for use by other users.

The **get s.lang** command creates **lang** as read-only and keeps no information regarding its creation. Because you are going to make changes to it, **get** must be informed of your intention to do so. This is done as follows:

get –e s.lang

get –e causes SCCS to create **lang** for both reading and writing (editing). It also places certain information about **lang** in another new file, called the "p.file" (**p.lang** in this case), which is needed later by the **delta** command.

get –e prints the same messages as **get**, except that now the SID for the first delta you will create is issued:

```
1.1
new delta   1.2
5 lines
```

Change **lang** by adding two more programming languages:

```
SNOBOL
ADA
```

Recording Changes by means of delta

Next, use the **delta** command as follows:

delta s.lang

delta then prompts with

comments?

Your response should be an explanation of why the changes were made. For example,

added more languages

delta now reads the p.file, **p.lang**, and determines what changes you made to **lang**. It does this by doing its own **get** to retrieve the original version and applying the **diff(1)** command, documented in the *User's/System Administrator's Reference Manual, to the original version and the edited version.* Next, **delta** stores the changes in **s.lang** and destroys the no longer needed **p.lang** and **lang** files.

When this process is complete, **delta** outputs

```
1.2
2 inserted
0 deleted
5 unchanged
```

The number 1.2 is the SID of the delta you just created, and the next three lines summarize what was done to **s.lang**.

Additional Information about get

The command,

get s.lang

retrieves the latest version of the file **s.lang**, now 1.2. SCCS does this by starting with the original version of the file and applying the delta you made. If you use the **get** command now, any of the following will retrieve version 1.2.

get s.lang
get –r1 s.lang
get –r1.2 s.lang

The numbers following **–r** are SIDs. When you omit the level number of the SID (as in **get –r1 s.lang**), the default is the highest level number that exists within the specified release. Thus, the second command requests the retrieval of the latest version in release 1, namely 1.2. The third command specifically requests the retrieval of a particular version, in this case also 1.2.

Whenever a major change is made to a file, you may want to signify it by changing the release number, the first number of the SID. This, too, is done with the **get** command.

get –e –r2 s.lang

Because release 2 does not exist, **get** retrieves the latest version before release 2. **get** also interprets this as a request to change the release number of the new delta to 2, thereby naming it 2.1 rather than 1.3. The output is

```
1.2
new delta 2.1
7 lines
```

which means version 1.2 has been retrieved, and 2.1 is the version **delta** will create. If the file is now edited, for example, by deleting COBOL from the list of languages, and **delta** is executed

> **delta s.lang**
> comments? **deleted cobol from list of languages**

you will see by **delta**'s output that version 2.1 is indeed created.

```
2.1
0 inserted
1 deleted
6 unchanged
```

Deltas can now be created in release 2 (deltas 2.2, 2.3, etc.), or another new release can be created in a similar manner.

The help **Command**

If the command

> **get lang**

is now executed, it will generate the following message:

```
ERROR [lang]: not an SCCS file (co1)
```

The code **co1** can be used with **help** to print a fuller explanation of the message.

> **help co1**

This gives the following explanation of why **get lang** produced an error message:

```
co1:
"not an SCCS file"
A file that you think is an SCCS file
does not begin with the characters "s.".
```

help is useful whenever there is doubt about the meaning of almost any SCCS message.

Delta Numbering

Think of deltas as the nodes of a tree in which the root node is the original version of the file. The root is normally named 1.1 and deltas (nodes) are named 1.2, 1.3, etc. The components of these SIDs are called release and level numbers, respectively. Thus, normal naming of new deltas proceeds by incrementing the level number. This is done automatically by SCCS whenever a delta is made.

Because the user may change the release number to indicate a major change, the release number then applies to all new deltas unless specifically changed again. Thus, the evolution of a particular file could be represented by Figure 14-1.
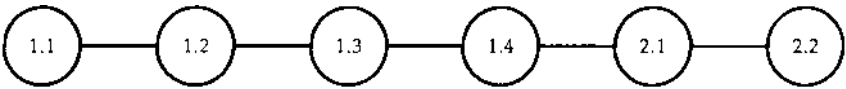
Figure 14-1: Evolution of an SCCS File

This is the normal sequential development of an SCCS file, with each delta dependent on the preceding deltas. Such a structure is called the trunk of an SCCS tree.

There are situations that require branching an SCCS tree. That is, changes are planned to a given delta that will not be dependent on all previous deltas. For example, consider a program in production use at version 1.3 and for which development work on release 2 is already in progress. Release 2 may already have a delta in progress as shown in Figure 14-1. Assume that a production user reports a problem in version 1.3 that cannot wait to be repaired in release 2. The changes necessary to repair the trouble will be applied as a delta to version 1.3 (the version in production use). This creates a new version that will then be released to the user but will not affect the changes being applied for release 2 (i.e., deltas 1.4, 2.1, 2.2, etc.). This new delta is the first node of a new branch of the tree.

Branch delta names always have four SID components: the same release number and level number as the trunk delta, plus a branch number and sequence number. The format is as follows:

release.level.branch.sequence

The branch number of the first delta branching off any trunk delta is always 1, and its sequence number is also 1. For example, the full SID for a delta branching off trunk delta 1.3 will be 1.3.1.1. As other deltas on that same branch are created, only the sequence number changes: 1.3.1.2, 1.3.1.3, etc. This is shown in Figure 14-2.
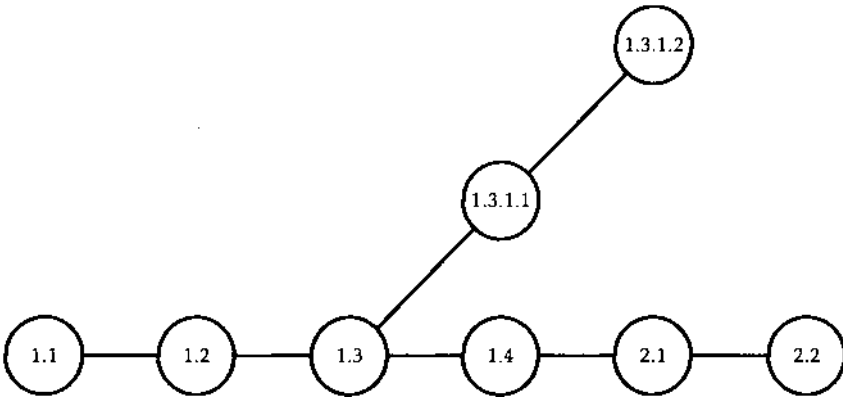
Figure 14-2: Tree Structure with Branch Deltas

The branch number is incremented only when a delta is created that starts a new branch off an existing branch, as shown in Figure 14-3. As this secondary branch develops, the sequence numbers of its deltas are incremented (1.3.2.1, 1.3.2.2, etc.), but the secondary branch number remains the same.
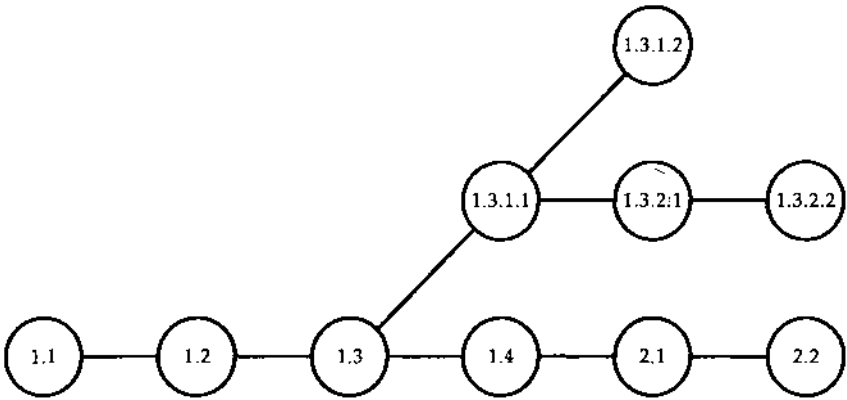
Figure 14-3: Extended Branching Concept

The concept of branching may be extended to any delta in the tree, and the numbering of the resulting deltas proceeds as shown above. SCCS allows the generation of complex tree structures. Although this capability has been provided for certain specialized uses, the SCCS tree should be kept as simple as possible. Comprehension of its structure becomes difficult as the tree becomes complex.

SCCS Command Conventions

SCCS commands accept two types of arguments:

- key letters
- file names

Key letters are options that begin with a minus sign, -, followed by a lowercase letter and, in some cases, a value.

File and/or directory names specify the file(s) the command is to process. Naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files and unreadable files [because of permission modes by means of **chmod**(1), documented in the *User's/System Administrator's Reference Manual*] in the named directories are silently ignored.

In general, file name arguments may not begin with a minus sign. If a file name of - (a lone minus sign) is specified, the command will read the standard input (usually your terminal) for lines and take each line as the name of an SCCS file to be processed. The standard input is read until end-of-file. This feature is often used in pipelines with, for example, the commands **find**(1) or **ls**(1), which are also documented in the *User's/System Administrator's Reference Manual*.

Key letters are processed before file names. Therefore, the placement of key letters is arbitrary—that is, they may be interspersed with file names. File names, however, are processed left to right. Somewhat different conventions apply to **what**(1), **sccsdiff**(1), and **val**(1), detailed later under "SCCS Commands" and documented in the *Programmer's Reference Manual*.

Certain actions of various SCCS commands are controlled by flags appearing in SCCS files. Some of these flags will be discussed, but for a complete description see **admin**(1) in the *Programmer's Reference Manual*.

The distinction between real user [see **passwd**(1)] and effective user will be of concern in discussing various actions of SCCS commands. For now, assume that the real and effective users are the same—the person logged into the UNIX System.

x.files and z.files

All SCCS commands that modify an SCCS file do so by writing a copy called the "x.file." This is done to ensure that the SCCS file is not damaged if processing terminates abnormally. SCCS names the x.file by replacing the **s.** of the SCCS file name with **x.**. The x.file is created in the same directory as the SCCS file, given the same mode [see **chmod**(1) in the *User's/System Administrator's Reference Manual*], and is owned by the effective user. When processing is complete, the old SCCS file is destroyed and the modified x.file is renamed (**x.** is relaced by **s.**) and becomes the new SCCS file.

To prevent simultaneous updates to an SCCS file, the same modifying commands also create a lock-file called the "z.file." SCCS forms its name by replacing the **s.** of the SCCS file name with a **z.** prefix. The z.file contains the process number of the command that creates it, and its existence prevents other commands from processing the SCCS file. The z.file is created with access permission mode 444 (read only) in the same directory as the SCCS file and is owned by the effective user. It exists only for the duration of the execution of the command that creates it.

In general, users can ignore x.files and z.files. They are useful only in the event of system crashes or similar situations.

Error Messages

SCCS commands produce error messages on the diagnostic output in this format:

        ERROR [name-of-file-being-processed]: message text (code)

The code in parentheses can be used as an argument to the **help** command to obtain a further explanation of the message. Detection of a fatal error during the processing of a file causes the SCCS command to stop processing that file and proceed with the next file specified.

SCCS Commands

This section describes the major features of the fourteen SCCS commands and their most common arguments. Full descriptions with details of all arguments are in the *Programmer's Reference Manual*.

Here is a quick-reference overview of the commands:

get retrieves versions of SCCS files

unget undoes the effect of a **get –e** prior to the file being **delta**ed

delta applies deltas (changes) to SCCS files and creates new versions

admin initializes SCCS files, manipulates their descriptive text, and controls delta creation rights

prs prints portions of an SCCS file in user-specified format

sact prints information about files that are currently out for edit

rmdel removes a delta from an SCCS file; allows removal of deltas created by mistake

cdc changes the commentary associated with a delta

what searches any UNIX System file(s) for all occurrences of a special pattern and prints out what follows it; useful in finding identifying information inserted by the **get** command

sccsdiff shows differences between any two versions of an SCCS file

comb combines consecutive deltas into one to reduce the size of an SCCS file

val validates an SCCS file

vc a filter that may be used for version control

The get **Command**

The **get**(1) command creates a file that contains a specified version of an SCCS file. The version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The resulting file is called the "g.file." It is created in the current directory and is owned by the real user. The mode assigned to the g.file depends on how the

get command is used.

The most common use of **get** is

    get s.abc

which normally retrieves the latest version of file **abc** from the SCCS file tree trunk and produces (for example) on the standard output

```
1.3
67 lines
No id keywords (cm7)
```

meaning version 1.3 of file **s.abc** was retrieved (assuming 1.3 is the latest trunk delta), it has 67 lines of text, and no ID keywords were substituted in the file.

The generated g.file (file **abc**) is given access permission mode 444 (read only). This particular way of using **get** is intended to produce g.files only for inspection, compilation, etc. It is not intended for editing (making deltas).

When several files are specified, the same information is output for each one. For example,

    get s.abc s.xyz

produces

```
s.abc:
1.3
67 lines
No id keywords (cm7)

s.xyz:
1.7
85 lines
No id keywords (cm7)
```

ID Keywords

In generating a g.file for compilation, it is useful to record the date and time of creation, the version retrieved, the module's name, etc., within the g.file. This information appears in a load module when one is eventually created. SCCS provides a convenient mechanism for doing this automatically. Identification (ID) key words appearing anywhere in the generated file are replaced by appropriate values according to the definitions of those ID key

words. The format of an ID keyword is an uppercase letter enclosed by percent signs, %. For example,

%I%

is the ID key word replaced by the SID of the retrieved version of a file. Similarly, **%H%** and **%M%** are the names of the g.file. Thus, executing **get** on an SCCS file that contains the PL/I declaration,

DCL ID CHAR(100) VAR INIT('%M% %I% %H%');

gives (for example) the following:

DCL ID CHAR(100) VAR INIT('MODNAME 2.3 07/18/85');

When no ID key words are substituted by **get**, the following message is issued:

No id keywords (cm7)

This message is normally treated as a warning by **get**, although the presence of the **i** flag in the SCCS file causes it to be treated as an error. For a complete list of the approximately twenty ID key words provided, see **get**(1) in the *Programmer's Reference Manual*.

Retrieval of Different Versions

The version of an SCCS file **get** retrieves is the most recently created delta of the highest numbered trunk release. However, any other version can be retrieved with **get –r** by specifying the version's SID. Thus,

get –r1.3 s.abc

retrieves version 1.3 of file **s.abc** and produces (for example) on the standard output

1.3
64 lines

A branch delta may be retrieved similarly,

get –r1.5.2.3 s.abc

which produces (for example) on the standard output

1.5.2.3
234 lines

When a SID is specified and the particular version does not exist in the SCCS

file, an error message results.

Omitting the level number, as in

get -r3 s.abc

causes retrieval of the trunk delta with the highest level number within the given release. Thus, the above command might output,

```
3.7
213 lines
```

If the given release does not exist, **get** retrieves the trunk delta with the highest level number within the highest-numbered existing release that is lower than the given release. For example, assume release 9 does not exist in file **s.abc** and release 7 is the highest-numbered release below 9. Executing

get -r9 s.abc

might produce

```
7.6
420 lines
```

which indicates that trunk delta 7.6 is the latest version of file **s.abc** below release 9. Similarly, omitting the sequence number, as in

get -r4.3.2 s.abc

results in the retrieval of the branch delta with the highest sequence number on the given branch. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8
89 lines
```

get -t will retrieve the latest (top) version of a particular release when no -r is used or when its value is simply a release number. The latest version is the delta produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

get -r3 -t s.abc

might produce

```
3.5
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce

```
3.2.1.5
46 lines
```

Retrieval With Intent to Make a Delta

The **get** –e command indicates an intent to make a delta. First, **get** checks the following conditions.

1. It checks if the login name or group ID of the person executing **get** is present in the user list. The login name or group ID must be present for the user to be allowed to make deltas. (See "The **admin** Command" for a discussion of making user lists.)

2. It checks if the release number (R) of the version being retrieved satisfies the relation

    > floor is less than or equal to R,
    > which is less than or equal to ceiling.

    This check determines if the release being accessed is a protected release. The floor and ceiling are flags in the SCCS file representing start and end of range.

3. It checks if the R is not locked against editing. The lock is a flag in the SCCS file.

4. It checks if multiple concurrent edits are allowed for the SCCS file by the **j** flag in the SCCS file.

A failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, **get** –e causes the creation of a g.file in the current directory with mode 644 (readable by everyone, writable only by the owner) owned by the real user. If a writable g.file already exists, **get** terminates with an error. This is to prevent inadvertent destruction of a g.file being edited for the purpose of making a delta.

Any ID keywords appearing in the g.file are not substituted by **get** –e because the generated g.file is subsequently used to create another delta. Replacement of ID keywords causes them to be permanently changed in the SCCS file. Because of this, **get** does not need to check for their presence in the g.file. Thus, the message

```
No id keywords (cm7)
```

is never output when **get -e** is used.

In addition, **get -e** causes the creation (or updating) of a p.file that is used to pass information to the **delta** command.

The following

    get -e s.abc

produces (for example) on the standard output

```
1.3
new delta 1.4
67 lines
```

Undoing a get -e

There may be times when a file is retrieved for editing in error; there is really no editing that needs to be done at this time. In such cases, the **unget** command can be used to cancel the delta reservation that was set up.

Additional get Options

If **get -r** and/or **-t** are used together with **-e**, the version retrieved for editing is the one specified with **-r** and/or **-t**.

get -i and **-x** are used to specify a list [see **get**(1) in the *Programmer's Reference Manual* for the syntax of such a list] of deltas to be included and excluded, respectively. Including a delta means forcing its changes to be included in the retrieved version. This is useful in applying the same changes to more than one version of the SCCS file. Excluding a delta means forcing it not to be applied. This may be used to undo the effects of a previous delta in the version to be created.

Whenever deltas are included or excluded, **get** checks for possible interference with other deltas. Two deltas can interfere, for example, when each one changes the same line of the retrieved g.file. A warning shows the range of lines within the retrieved g.file where the problem may exist. The user should examine the g.file to determine what the problem is and take appropriate corrective steps (e.g., edit the file).
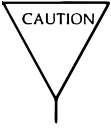
 get –i and **get –x** should be used with extreme care.

get –k is used either to regenerate a g.file that may have been accidentally removed or ruined after **get –e**, or simply to generate a g.file in which the replacement of ID keywords has been suppressed. A g.file generated by **get –k** is identical to one produced by **get –e**, but no processing related to the p.file takes place.

Concurrent Edits of Different SID

The ability to retrieve different versions of an SCCS file allows several deltas to be in progress at any given time. This means that several **get –e** commands may be executed on the same file as long as no two executions retrieve the same version (unless multiple concurrent edits are allowed).

The p.file created by **get –e** is named by automatic replacement of the SCCS file name's prefix **s.** with **p.**. It is created in the same directory as the SCCS file, given mode 644 (readable by everyone, writable only by the owner), and owned by the effective user. The p.file contains the following information for each delta that is still in progress:

- the SID of the retrieved version

- the SID given to the new delta when it is created

- the login name of the real user executing **get**

The first execution of **get –e** causes the creation of a p.file for the corresponding SCCS file. Subsequent executions only update the p.file with a line containing the above information. Before updating, however, **get** checks to assure that no entry already in the p.file specifies that the SID of the version to be retrieved is already retrieved (unless multiple concurrent edits are allowed). If the check succeeds, the user is informed that other deltas are in progress and processing continues. If the check fails, an error message results.

It should be noted that concurrent executions of **get** must be carried out from different directories. Subsequent executions from the same directory will attempt to overwrite the g.file, which is an SCCS error condition. In practice, this problem does not arise since each user normally has a different working directory. See "Protection" under "SCCS Files" for a discussion of how

different users are permitted to use SCCS commands on the same files.

Figure 14-4 shows the possible SID components a user can specify with **get** (left-most column), the version that will then be retrieved by **get**, and the resulting SID for the delta, which **delta** will create (right-most column).

SID Specified in get*	-b Key-Letter Used†	Other Conditions	SID Retrieved by get	SID of Delta To be Created by delta
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1)
R	no	R > mR	mR.mL	R.1§
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	–	R< mR and R does not exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk successor number in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L.	no	No trunk successor	R.L	R.(L+1)
R.L.	yes	No trunk successor	R.L	R.L.(mB+1).1
R.L	–	Trunk successor in release ≥ R	R.L	R.L.(mS+1).1
R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch successor	R.L.B.S	R.L.(mB+1).1

Footnotes *, †, ‡, §, and ** on next page.

Figure 14-4: Determination of New SID

Footnotes to Figure 14-4:

*   R, L, B, and S mean release, level, branch, and sequence numbers in the SID, and m means maximum. Thus, for example, R.mL means the maximum level number within release R. R.L.(mB+1).1 means the first sequence number on the new branch (i.e., maximum branch number plus 1) of level L within release R. Note that if the SID specified is R.L, R.L.B, or R.L.B.S, each of these specified SID numbers must exist.

†   The –b key letter is effective only if the **b** flag [see **admin**(1) in the *Programmer's Reference Manual*] is present in the file. An entry of – means irrelevant.

‡   This case applies if the **d** (default SID) flag is not present. If the **d** flag is present in the file, the SID is interpreted as if specified on the command line. Thus, one of the other cases in this figure applies.

§   This is used to force the creation of the first delta in a new release.

**   hR is the highest existing release that is lower than the specified, nonexistent release R.

Concurrent Edits of Same SID

Under normal conditions, more than one **get –e** for the same SID is not permitted. That is, **delta** must be executed before a subsequent **get –e** is executed on the same SID.

Multiple concurrent edits are allowed if the **j** flag is set in the SCCS file. Thus,

```
get -e s.abc
1.1
new delta 1.2
5 lines
```

may be immediately followed by

```
get -e s.abc
1.1
new delta 1.1.1.1
5 lines
```

without an intervening **delta**. In this case, a **delta** after the first **get** will produce delta 1.2 (assuming 1.1 is the most recent trunk delta), and a **delta** after

the second **get** will produce delta 1.1.1.1.

Key Letters That Affect Output

get -p causes the retrieved text to be written to the standard output rather than to a g.file. In addition, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the diagnostic output. **get -p** is used, for example, to create a g.file with an arbitrary name, as in

> **get -p s.abc** > *arbitrary-file-name*

get -s suppresses output normally directed to the standard output, such as the SID of the retrieved version and the number of lines retrieved, but it does not affect messages normally directed to the diagnostic output. **get -s** is used to prevent nondiagnostic messages from appearing on the user's terminal and is often used with **-p** to pipe the output, as in

> **get -p -s s.abc | pg**

get -g suppresses the retrieval of the text of an SCCS file. This is useful in several ways. For example, to verify a particular SID in an SCCS file

> **get -g -r4.3 s.abc**

outputs the SID 4.3 if it exists in the SCCS file **s.abc** or an error message if it does not. Another use of **get -g** is in regenerating a p.file that may have been accidentally destroyed, as in

> **get -e -g s.abc**

get -l causes SCCS to create an "l.file." It is named by replacing the **s.** of the SCCS file name with **l.**, created in the current directory with mode 444 (read only) and owned by the real user. The l.file contains a table [whose format is described under **get**(1) in the *Programmer's Reference Manual*] showing the deltas used in constructing a particular version of the SCCS file. For example

> **get -r2.3 -l s.abc**

generates an l.file showing the deltas applied to retrieve version 2.3 of file **s.abc**. Specifying **p** with **-l**, as in

> **get -lp -r2.3 s.abc**

causes the output to be written to the standard output rather than to the l.file.

get -g can be used with **-l** to suppress the retrieval of the text.

get -m identifies the changes applied to an SCCS file. Each line of the g.file is preceded by the SID of the delta that caused the line to be inserted. The SID is separated from the text of the line by a tab character.

get -n causes each line of a g.file to be preceded by the value of the ID keyword and a tab character. This is most often used in a pipeline with **grep**(1), which is documented in the *User's/System Administrator's Reference Manual*. For example, to find all lines that match a given pattern in the latest version of each SCCS file in a directory, the following may be executed:

    get -p -n -s *directory* | **grep** *pattern*

If both **-m** and **-n** are specified, each line of the generated g.file is preceded by the value of the **chap3.13** ID keyword and a tab (this is the effect of **-n**) and is followed by the line in the format produced by **-m**. Because use of **-m** and/or **-n** causes the contents of the g.file to be modified, such a g.file must not be used for creating a delta. Therefore, neither **-m** nor **-n** may be specified together with **get -e**.

| NOTE | See **get**(1) in the *Programmer's Reference Manual* for a full description of additional key letters. |

The delta **Command**

The **delta**(1) command is used to incorporate changes made to a g.file into the corresponding SCCS file—that is, to create a delta and, therefore, a new version of the file.

The **delta** command requires the existence of a p.file (created by means of **get -e**). It examines the p.file to verify the presence of an entry containing the user's login name. If none is found, an error message results.

The **delta** command performs the same permission checks that **get -e** performs. If all checks are successful, **delta** determines what has been changed in the g.file by comparing it by means of **diff**(1), documented in the *User's/System Administrator's Reference Manual*, with its own temporary copy of the g.file as it was before editing. This temporary copy of the g.file is called the d.file and is obtained by performing an internal **get** on the SID

specified in the p.file entry.

The required p.file entry is the one containing the login name of the user executing **delta**, because the user who retrieved the g.file must be the one who creates the delta. However, if the login name of the user appears in more than one entry, the same user has executed **get -e** more than once on the same SCCS file. Then, **delta -r** must be used to specify the SID that uniquely identifies the p.file entry. This entry is then the one used to obtain the SID of the delta to be created.

In practice, the most common use of **delta** is

>   **delta s.abc**

which prompts

>   comments?

to which the user replies with a description of why the delta is being made, ending the reply with a new-line character. The user's response may be up to 512 characters long with new-lines (not intended to terminate the response) escaped by backslashes, \.

If the SCCS file has a **v** flag, **delta** first prompts with

>   MRs?

(Modification Requests), on the standard output. The standard input is then read for MR numbers, separated by blanks and/or tabs, ended with a new-line character. A Modification Request is a formal way of asking for a correction or enhancement to the file. In some controlled environments where changes to source files are tracked, deltas are permitted only when initiated by a trouble report, change request, trouble ticket, etc., collectively called MRs. Recording MR numbers within deltas is a way of enforcing the rules of the change management process.

delta -y and/or **-m** can be used to enter comments and MR numbers on the command line rather than through the standard input, as in

>   **delta -y** "*descriptive comment*" **-m** "*mrnum1 mrnum2*" **s.abc**

In this case, the prompts for comments and MRs are not printed, and the standard input is not read. These two key letters are useful when **delta** is executed from within a shell procedure [see **sh**(1) in the *User's/System Administrator's Reference Manual*].

> | NOTE | **delta –m** is allowed only if the SCCS file has a **v** flag.

No matter how comments and MR numbers are entered with **delta**, they are recorded as part of the entry for the delta being created. Also, they apply to all SCCS files specified with the **delta**.

If **delta** is used with more than one file argument and the first file named has a **v** flag, all files named must have this flag. Similarly, if the first file named does not have the flag, none of the files named may have it.

When **delta** processing is complete, the standard output displays the SID of the new delta (from the p.file) and the number of lines inserted, deleted, and left unchanged. For example:

```
1.4
14 inserted
7 deleted
345 unchanged
```

If line counts do not agree with the user's perception of the changes made to a g.file, it may be because there are various ways to describe a set of changes, especially if lines are moved around in the g.file. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should always agree with the number of lines in the edited g.file.

If you are in the process of making a delta and the **delta** command finds no ID keywords in the edited g.file, the message

```
No id keywords (cm7)
```

is issued after the prompts for commentary but before any other output. This means that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. This could be caused by making a delta from a g.file that was created by a **get** without –e (ID keywords are replaced by **get** in such a case). It could also be caused by accidentally deleting or changing ID keywords while editing the g.file. Or, it is possible that the file had no ID keywords. In any case, the delta will be created unless there is an **i** flag in the SCCS file (meaning the error should be treated as fatal), in which case the delta will not be created.

After the processing of an SCCS file is complete, the corresponding p.file entry is removed from the p.file. All updates to the p.file are made to a temporary copy, the "q.file," whose use is similar to the use of the x.file described earlier under "SCCS Command Conventions." If there is only one entry in the p.file, then the p.file itself is removed.

In addition, **delta** removes the edited g.file unless **–n** is specified. For example

> **delta –n s.abc**

will keep the g.file after processing.

delta –s suppresses all output normally directed to the standard output, other than comments? and MRs?. Thus, use of **–s** with **–y** (and/or **–m**) causes **delta** to neither read the standard input nor write the standard output.

The differences between the g.file and the d.file constitute the delta and may be printed on the standard output by using **delta –p**. The format of this output is similar to that produced by **diff**(1), documented in the *User's/System Administrator's Reference Manual*.

The admin **Command**

The **admin**(1) command, documented in the *Programmer's Reference Manual*, is used to administer SCCS files—that is, to create new SCCS files and change the parameters of existing ones. When an SCCS file is created, its parameters are initialized by use of key letters with **admin** or are assigned default values if no key letters are supplied. The same key letters are used to change the parameters of existing SCCS files.

Two key letters are used in detecting and correcting corrupted SCCS files (see "Auditing" under "SCCS Files").

Newly created SCCS files are given access permission mode 444 (read only) and are owned by the effective user. Only a user with write permission in the directory containing the SCCS file may use the **admin** command on that file.

Creation of SCCS Files

An SCCS file can be created by executing the command

admin –ifirst s.abc

in which the value **first** with **–i** is the name of a file from which the text of the initial delta of the SCCS file **s.abc** is to be taken. Omission of a value with **–i** means **admin** is to read the standard input for the text of the initial delta.

The command

admin –i s.abc < first

is equivalent to the previous example.

If the text of the initial delta does not contain ID keywords, the message

No id keywords (cm7)

is issued by **admin** as a warning. However, if the command also sets the **i** flag (not to be confused with the **–i** key letter), the message is treated as an error and the SCCS file is not created. Only one SCCS file may be created at a time using **admin –i**.

admin –r is used to specify a release number for the first delta. Thus:

admin –ifirst –r3 s.abc

means the first delta should be named 3.1 rather than the normal 1.1. Because **–r** has meaning only when creating the first delta, its use is permitted only with **–i**.

Inserting Commentary for the Initial Delta

When an SCCS file is created, the user may want to record why this was done. Comments (**admin –y**) and/or MR numbers (**–m**) can be entered in exactly the same way as a **delta**.

If **–y** is omitted, a comment line of the form

date and time created YY/MM/DD HH:MM:SS by logname

is automatically generated.

If it is desired to supply MR numbers (**admin -m**), the **v** flag must be set by means of **-f**. The **v** flag simply determines whether MR numbers must be supplied when using any SCCS command that modifies a delta commentary [see **sccsfile**(4) in the *Programmer's Reference Manual*] in the SCCS file. Thus:

> **admin -ifirst -m**mrnum1 **-fv s.abc**

Note that **-y** and **-m** are effective only if a new SCCS file is being created.

Initialization and Modification of SCCS File Parameters

Part of an SCCS file is reserved for descriptive text, usually a summary of the file's contents and purpose. It can be initialized or changed by using **admin -t**.

When an SCCS file is first being created and **-t** is used, it must be followed by the name of a file from which the descriptive text is to be taken. For example, the command

> **admin -ifirst -tdesc s.abc**

specifies that the descriptive text is to be taken from file **desc**.

When processing an existing SCCS file, **-t** specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus:

> **admin -tdesc s.abc**

specifies that the descriptive text of the SCCS file is to be replaced by the contents of **desc**. Omission of the file name after the **-t** key letter as in

> **admin -t s.abc**

causes the removal of the descriptive text from the SCCS file.

The flags of an SCCS file may be initialized or changed by **admin -f** or deleted by means of **-d**.

SCCS file flags are used to direct certain actions of the various commands. [See **admin**(1) in the *Programmer's Reference Manual* for a description of all the flags.] For example, the **i** flag specifies that a warning message (stating that there are no ID keywords contained in the SCCS file) should be treated as an error. The **d** (default SID) flag specifies the default version of the SCCS file to be retrieved by the **get** command.

admin −f is used to set flags and, if desired, their values. For example

admin −ifirst −fi −fm*modname* **s.abc**

sets the **i** and **m** (module name) flags. The value *modname* specified for the **m** flag is the value that the **get** command will use to replace the %M% ID keyword. (In the absence of the **m** flag, the name of the g.file is used as the replacement for the %M% ID keyword.) Several −**f** key letters may be supplied on a single **admin**, and they may be used whether the command is creating a new SCCS file or processing an existing one.

admin −d is used to delete a flag from an existing SCCS file. As an example, the command

admin −dm s.abc

removes the **m** flag from the SCCS file. Several −**d** key letters may be used with one **admin** and may be intermixed with −**f**.

SCCS files contain a list of login names and/or group IDs of users who are allowed to create deltas. This list is empty by default, allowing anyone to create deltas. To create a user list (or add to an existing one), **admin −a** is used. For example,

admin −axyz −awql −a1234 s.abc

adds the login names **xyz** and **wql** and the group ID **1234** to the list. **admin −a** may be used whether creating a new SCCS file or processing an existing one.

admin −e (erase) is used to remove login names or group IDs from the list.

The prs **Command**

The **prs**(1) command is used to print all or part of an SCCS file on the standard output. If **prs −d** is used, the output will be in a format called data specification. Data specification is a string of SCCS file data key words (not to be confused with **get** ID keywords) interspersed with optional user text.

Data key words are replaced by appropriate values according to their definitions. For example,

:I:

is defined as the data key word replaced by the SID of a specified delta.

Similarly, **:F:** is the data key word for the SCCS file name currently being processed, and **:C:** is the comment line associated with a specified delta. All parts of an SCCS file have an associated data key word. For a complete list, see **prs**(1) in the *Programmer's Reference Manual*.

There is no limit to the number of times a data key word may appear in a data specification. Thus, for example,

> **prs –d":I: this is the top delta for :F: :I: " s.abc**

may produce on the standard output

> 2.1 this is the top delta for s.abc 2.1

Information may be obtained from a single delta by specifying its SID using **prs –r**. For example,

> **prs –d":F:: :I: comment line is: :C: " –r1.4 s.abc**

may produce the following output:

> s.abc: **1.4 comment line is:** THIS IS A COMMENT

If **–r** is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained with **–l** or **–e**. The use of **prs –e** substitutes data keywords for the SID designated by means of **–r** and all deltas created earlier, while **prs –l** substitutes data keywords for the SID designated by means of **–r** and all deltas created later. Thus, the command

> **prs –d:I: –r1.4 –e s.abc**

may output

> 1.4
> 1.3
> 1.2.1.1
> 1.2
> 1.1

and the command

> **prs –d:I: –r1.4 –l s.abc**

may produce

```
3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4
```

Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both –e and –l.

The sact **Command**

sact(1) is like a special form of the **prs** command that produces a report about files that are out for edit. The command takes only one type of argument: a list of file or directory names. The report shows the SID of any file in the list that is out for edit, the SID of the impending delta, the login of the user who executed the **get –e** command, and the date and time the **get –e** was executed. It is a useful command for an administrator. It is described in the *Programmer's Reference Manual*.

The rmdel **Command**

The **rmdel**(1) command, documented in the *Programmer's Reference Manual*, allows removal of a delta from an SCCS file. Its use should be reserved for deltas in which incorrect global changes were made. The delta to be removed must be a leaf delta. That is, it must be the most recently created delta on its branch or on the trunk of the SCCS file tree. In Figure 14-3, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed. Only after they are removed can deltas 1.3.2.1 and 2.1 be removed.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must be either the one who created the delta being removed or the owner of the SCCS file and its directory.

The −r key letter is mandatory with **rmdel**. It is used to specify the complete SID of the delta to be removed. Thus,

>   **rmdel −r2.3 s.abc**

specifies the removal of trunk delta 2.3.

Before removing the delta, **rmdel** checks that the release number (R) of the given SID satisfies the relation:

>   floor less than or equal to R less than or equal to ceiling

The **rmdel** command also checks the SID to make sure it is not for a version on which a **get** for editing has been executed and whose associated **delta** has not yet been made. In addition, the login name or group ID of the user must appear in the file's user list (or the user list must be empty). Also, the release specified cannot be locked against editing. That is, if the l flag is set [see **admin**(1) in the *Programmer's Reference Manual*], the release must not be contained in the list. If these conditions are not satisfied, processing is terminated, and the delta is not removed.

Once a specified delta has been removed, its type indicator in the delta table of the SCCS file is changed from D (delta) to R (removed).

The cdc **Command**

The **cdc**(1) command, documented in the *Programmer's Reference Manual*, is used to change the commentary made when the delta was created. It is similar to the **rmdel** command (e.g., −r and full SID are necessary), although the delta need not be a leaf delta. For example,

>   **cdc −r3.4 s.abc**

specifies that the commentary of delta 3.4 is to be changed. New commentary is then prompted for as with **delta**.

The old commentary is kept, but it is preceded by a comment line indicating that it has been superseded, and the new commentary is entered ahead of the comment line. The inserted comment line records the login name of the user executing **cdc** and the time of its execution.

The **cdc** command also allows for the insertion of new and deletion of old ("!" prefix) MR numbers. Thus,

> cdc –r1.4 s.abc
>
> | MRs? mrnum3 !mrnum1 | **(The MRs? prompt appears only if the v flag has been set.)** |
> |---|---|
> | comments? | **deleted wrong MR number and inserted correct MR number** |

inserts **mrnum3** and deletes **mrnum1** for delta 1.4.

NOTE	An MR (Modification Request) is described in "The **delta** Command" section.

The what **Command**

The **what**(1) command, described in the *Programmer's Reference Manual*, is used to find identifying information within any UNIX System file whose name is given as an argument. No key letters are accepted. The **what** command searches the given file(s) for all occurrences of the string **@(#)**, which is the replacement for the **%Z%** ID keyword [see **get**(1) in the *Programmer's Reference Manual*]. It prints on the standard output whatever follows the string until the first double quote, ", greater than, >, backslash, \, new-line, or nonprinting NUL character.

For example, if an SCCS file called **s.prog.c** (a C language program) contains the following line:

> char   id[ ]= "%W%";

and the command

> get –r3.4 s.prog.c

is used, the resulting g.file is compiled to produce **prog.o** and **a.out**. Then, the command

> what prog.c prog.o a.out

produces

```
prog.c:
  prog.c: .3.4
prog.o:
  prog.c:  3.4
a.out:
  prog.c:  3.4
```

The string searched for by **what** need not be inserted by means of an ID key-word of **get**; it may be inserted in any convenient manner.

The sccsdiff **Command**

The **sccsdiff**(1) command, documented in the *Programmer's Reference Manual*, determines (and prints on the standard output) the differences between any two versions of an SCCS file. The versions to be compared are specified with **sccsdiff -r** in the same way as with **get -r**. SID numbers must be specified as the first two arguments. Any following key letters are inter-preted as arguments to the **pr**(1) command, documented in the *User's/System Administrator's Reference Manual*, (which prints the differences) and must appear before any file names. The SCCS file(s) to be processed are named last. Directory names and a name of - (a lone minus sign) are not acceptable to **sccsdiff**.

The following is an example of the format of **sccsdiff**:

    sccsdiff -r3.4 -r5.6 s.abc

The differences are printed the same way as by **diff**(1), which is in the *User's/System Administrator's Reference Manual*.

The comb **Command**

The **comb**(1) command, documented in the *Programmer's Reference Manual*, lets the user try to reduce the size of an SCCS file. It generates a shell procedure [see **sh**(1) in the *User's/System Administrator's Reference Manual*] on the standard output, which reconstructs the file by discarding unwanted deltas and combining other specified deltas. (It is not recom-mended that **comb** be used as a matter of routine.)

In the absence of any key letters, **comb** preserves only leaf deltas and the minimum number of ancestor deltas necessary to preserve the shape of an SCCS tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Thus, in Figure 14-3, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated.

Some of the key letters used with this command are as follows:

comb –s   This option generates a shell procedure that produces a report of the percentage space (if any) the user will save. This is often useful as an advance step.

comb –p   This option is used to specify the oldest delta the user wants preserved.

comb –c   This option is used to specify a list [see **get**(1) in the *Programmer's Reference Manual* for its syntax] of deltas the user wants preserved. All other deltas will be discarded.

The shell procedure generated by **comb** is not guaranteed to save space. A reconstructed file may even be larger than the original. Note, too, that the shape of an SCCS file tree may be altered by the reconstruction process.

The val **Command**

The **val**(1) command (see the *Programmer's Reference Manual*) is used to determine whether a file is an SCCS file meeting the characteristics specified by certain key letters. It checks for the existence of a particular delta when the SID for that delta is specified with **–r**.

The string following **–y** or **–m** is used to check the value set by the **t** or **m** flag, respectively. See **admin**(1) in the *Programmer's Reference Manual* for descriptions of these flags.

The **val** command treats the special argument – differently from other SCCS commands. It allows **val** to read the argument list from the standard input instead of from the command line, and the standard input is read until an end-of-file (CTRL-D) is entered. This permits one **val** command with different values for key letters and file arguments. For example,

val – –yc –mabc s.abc –mxyz –ypl1 s.xyz

first checks if file **s.abc** has a value **c** for its type flag and value **abc** for the module name flag. Once this is done, **val** processes the remaining file, in this case **s.xyz**.

The **val** command returns an 8-bit code. Each bit set shows a specific error [see **val**(1) for a description of errors and codes]. In addition, an appropriate diagnostic is printed unless suppressed by **-s**. A return code of 0 means all files met the characteristics specified.

The vc **Command**

The **vc**(1) command is an **awk**-like tool used for version control of sets of files. While it is distributed as part of the SCCS package, it does not require the files it operates on to be under SCCS control. A complete description of **vc** may be found in the *Programmer's Reference Manual*.

SCCS Files

This section covers protection mechanisms used by SCCS, the format of SCCS files, and the recommended procedures for auditing SCCS files.

Protection

SCCS relies on the capabilities of the UNIX System for most of the protection mechanisms required to prevent unauthorized changes to SCCS files—that is, changes by non-SCCS commands. Protection features provided directly by SCCS are the release lock flag, the release floor and ceiling flags, and the user list.

Files created by the **admin** command are given access permission mode 444 (read only). This mode should remain unchanged because it prevents modification of SCCS files by non-SCCS commands. Directories containing SCCS files should be given mode 755, which allows only the owner of the directory to modify it.

SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies their protection and auditing. The contents of directories should be logical groupings—subsystems of the same large project, for example.

SCCS files should have only one link (name) because commands that modify them do so by creating a copy of the file (the x.file; see "SCCS Command Conventions"). When processing is done, the old file is automatically removed and the x.file renamed (**s.** prefix). If the old file had additional links, this breaks them. Then, rather than process such files, SCCS commands will produce an error message.

When only one person uses SCCS, the real and effective user IDs are the same; and the user ID owns the directories containing SCCS files. Therefore, SCCS may be used directly without any preliminary preparation.

When several users with unique user IDs are assigned SCCS responsibilities (e.g., on large development projects), one user—that is, one user ID—must be chosen as the owner of the SCCS files. This person will administer the files (e.g., use the **admin** command) and will be SCCS administrator for the project. Because other users do not have the same privileges and permissions as the SCCS administrator, they are not able to execute directly those

commands that require write permission in the directory containing the SCCS files. Therefore, a project-dependent program is required to provide an interface to the **get, delta**, and, if desired, **rmdel** and **cdc** commands.

The interface program must be owned by the SCCS administrator and must have the set user ID on execution bit on [see **chmod**(1) in the *User's/System Administrator's Reference Manual*]. This assures that the effective user ID is the user ID of the SCCS administrator. With the privileges of the interface program during command execution, the owner of an SCCS file can modify it at will. Other users whose login names or group IDs are in the user list for that file (but are not the owner) are given the necessary permissions only for the duration of the execution of the interface program. Thus, they may modify SCCS only with **delta** and, possibly, **rmdel** and **cdc**.

A project-dependent interface program, as its name implies, can be custom built for each project.

Formatting

SCCS files are composed of lines of ASCII text arranged in six parts as follows:

Checksum   a line containing the logical sum of all the characters of the file (not including the checksum itself)

Delta Table   information about each delta, such as type, SID, date and time of creation, and commentary

User Names   list of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas

Flags   indicators that control certain actions of SCCS commands

Descriptive Text   usually a summary of the contents and purpose of the file

Body   the text administered by SCCS, intermixed with internal SCCS control lines

Details on these file sections may be found in **sccsfile**(4). The checksum is discussed below under "Auditing."

Since SCCS files are ASCII files they can be processed by non-SCCS commands like **ed**(1), **grep**(1), and **cat**(1), all documented in the *User's/System Administrator's Reference Manual*. This is convenient when an SCCS file must be modified manually (e.g., a delta's time and date were recorded incorrectly because the system clock was set incorrectly), or when a user wants simply to look at the file.

CAUTION
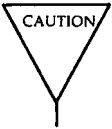
Extreme care should be exercised when modifying SCCS files with non-SCCS commands.

Auditing

When a system or hardware malfunction destroys an SCCS file, any command will issue an error message. Commands also use the checksum stored in an SCCS file to determine whether the file has been corrupted since it was last accessed [possibly by having lost one or more blocks or by having been modified with **ed**(1)]. No SCCS command will process a corrupted SCCS file except the **admin** command with **-h** or **-z**, as described below.

SCCS files should be audited for possible corruptions on a regular basis. The simplest and fastest way to do an audit is to use **admin -h** and specify all SCCS files:

      admin **-h** s.*file1* s.*file2* ...
                      or
      admin **-h** *directory1* *directory2* ...

If the new checksum of any file is not equal to the checksum in the first line of that file, the message

      corrupted file (co6)

is produced for that file. The process continues until all specified files have been examined. When examining directories (as in the second example above), the checksum process will not detect missing files. A simple way to learn whether files are missing from a directory is to execute the **ls**(1)

command, described in the *User's/System Administrator's Reference Manual*, periodically, and compare the outputs. Any file whose name appeared in a previous output but not in the current one no longer exists.

When a file has been corrupted, the way to restore it depends on the extent of the corruption. If damage is extensive, the best solution is to contact the local UNIX System operations group and request that the file be restored from a backup copy. If the damage is minor, repair through editing may be possible. After such a repair, the **admin** command must be executed:

> **admin –z** s.*file*

The purpose of this is to recompute the checksum and bring it into agreement with the contents of the file. After this command is executed, any corruption that existed in the file will no longer be detectable.

15 sdb—the Symbolic Debugger

Introduction

This chapter describes the symbolic debugger, **sdb**(1) in the *Programmer's Reference Manual*, as implemented for C language programs on the UNIX System V/386 Release 3.2 Operating System. The **sdb** program is useful both for examining core images of aborted programs and for providing an environment in which execution of a program can be monitored and controlled.

The **sdb** program allows interaction with a debugged program at the source language level. When debugging a core image from an aborted program, **sdb** reports which line in the source program caused the error and allows all variables to be accessed symbolically and to be displayed in the correct format.

When executing, breakpoints may be placed at selected statements, or the program may be single-stepped on a line-by-line basis. To facilitate specification of lines in the program without a source listing, **sdb** provides a mechanism for examining the source text. Procedures may be called directly from the debugger. This feature is useful both for testing individual procedures and for calling user-provided routines, which provide formatted printouts of structured data.

Using sdb

In order to use **sdb** to its full capabilities, it is necessary to compile the source program with the **-g** option. This causes the compiler to generate additional information about the variables and statements of the compiled program. When the **-g** option has been specified, **sdb** can be used to obtain a trace of the called functions at the time of the abort and interactively display the values of variables.

A typical sequence of shell commands for debugging a core image is

```
cc -g prgm.c -o prgm
prgm
Bus error - core dumped
sdb prgm
main:25:      x[i] = 0;
*
```

The program **prgm** was compiled with the **-g** option and then executed. An error occurred, which caused a core dump. The **sdb** program is then invoked to examine the core dump to determine the cause of the error. It reports that the bus error occurred in function **main** at line 25 (line numbers are always relative to the beginning of the file) and outputs the source text of the offending line. The **sdb** program then prompts the user with an *, which shows that it is waiting for a command.

It is useful to know that **sdb** has a notion of current function and current line. In this example, they are initially set to **main** and 25, respectively.

Here **sdb** was called with one argument, **prgm**. In general, it takes three arguments on the command line:

1. The first argument is the name of the executable file that is to be debugged. It defaults to **a.out** when not specified.

2. The second argument is the name of the core file, defaulting to **core**.

3. The third argument is the list of the directories (separated by colons) containing the source of the program being debugged. The default is the current working directory.

In the example, the second and third arguments defaulted to the correct values, so only the first was specified.

If the error occurred in a function that was not compiled with the –g option, **sdb** prints the function name and the address at which the error occurred. The current line and function are set to the first executable line in **main**. If **main** was not compiled with the –g option, **sdb** will print an error message, but debugging can continue for those routines that were compiled with the –g option.

Figure 15-1 at the end of the chapter, shows a more extensive example of **sdb** use.

Printing a Stack Trace

It is often useful to obtain a listing of the function calls that led to the error. This is obtained with the **t** command. For example:

```
*t
sub(x=2,y=3)        [prgm.c:25]
inter(i=16012)      [prgm.c:96]
main(argc=1,argv=0x7fffff54,envp=0x7fffff5c) [prgm.c: 15]
```

This indicates that the program was stopped within the function **sub** at line 25 in file **prgm.c**. The **sub** function was called with the arguments $x=2$ and $y=3$ from **inter** at line 96. The **inter** function was called from **main** at line 15. The **main** function is always called by a startup routine with three arguments often referred to as *argc*, *argv*, and *envp*. Note that *argv* and *envp* are pointers, so their values are printed in hexadecimal.

Examining Variables

The **sdb** program can be used to display variables in the stopped program. Variables are displayed by typing their name followed by a slash, so

```
*errflag/
```

causes **sdb** to display the value of variable **errflag**. Unless otherwise specified, variables are assumed to be either local to or accessible from the current function. To specify a different function, use the form

```
*sub:i/
```

to display variable **i** in function **sub**.

The **sdb** program supports a limited form of pattern matching for variable and function names. The symbol ∗ is used to match any sequence of characters of a variable name and **?** to match any single character. Consider the following commands

```
*x*/
*sub:y?/
**/
```

The first prints the values of all variables beginning with **x**, the second prints the values of all two-letter variables in function **sub** beginning with **y**, and the last prints all variables. In the first and last examples, only variables accessible from the current function are printed. The command

```
**:*/
```

displays the variables for each function on the call stack.

The **sdb** program normally displays the variable in a format determined by its type as declared in the source program. To request a different format, a specifier is placed after the slash. The specifier consists of an optional length specification followed by the format. The length specifiers are:

b one byte

h two bytes (half word)

l four bytes (long word)

The length specifiers are effective only with the formats **d, o, x**, and **u**. If no length is specified, the word length of the host machine is used. A number can be used with the **s** or **a** formats to control the number of characters printed. The **s** and **a** formats normally print characters until either a null is reached or 128 characters have been printed. The number specifies exactly how many characters should be printed.

There are a number of format specifiers available:

c character

d decimal

u decimal unsigned

o octal

x   hexadecimal

f   32-bit single-precision floating point

g   64-bit double-precision floating point

s   assume variable is a string pointer and print characters starting at the
     address pointed to by the variable until a null is reached

a   print characters starting at the variable's address until a null is reached

p   pointer to function

i   interpret as a machine-language instruction

For example, the variable **i** can be displayed with

>   *i/x

which prints out the value of **i** in hexadecimal.

   sdb also knows about structures, arrays, and pointers so that all of the fol-
lowing commands work.

>   *array[2][3]/
>   *sym.id/
>   *psym->usage/
>   *xsym[20].p->usage/

The only restriction is that array subscripts must be numbers. Note that as a
special case

>   *psym[0]

displays the structure pointed to by **psym** in decimal.

   Core locations can also be displayed by specifying their absolute
addresses. The command

>   *1024/

displays location 1024 in decimal. As in C language, numbers may also be
specified in octal or hexadecimal so the above command is equivalent to both

>   *02000/

and

>   *0x400/

It is possible to mix numbers and variables so that

> *1000.x/

refers to an element of a structure starting at address 1000, and

> *1000->x/

refers to an element of a structure whose address is at 1000. For commands of the type *1000.x/ and *1000->x/, the **sdb** program uses the structure template of the last structure referenced.

The address of a variable is printed with =, so

> *i=

displays the address of **i**. Another feature whose usefulness will become apparent later is the command

> *./

which redisplays the last variable typed.

Source File Display and Manipulation

The **sdb** program has been designed to make it easy to debug a program without constant reference to a current source listing. Facilities are provided that perform context searches within the source files of the program being debugged and that display selected portions of the source files. The commands are similar to those of the UNIX System text editor **ed**(1), documented in the *User's/System Administrator's Reference Manual*. Like the editor, **sdb** has a notion of current file and line within the current file. **sdb** also knows how the lines of a file are partitioned into functions, so it also has a notion of current function. As noted in other parts of this document, the current function is used by a number of **sdb** commands.

Displaying the Source File

Four commands exist for displaying lines in the source file. They are useful for perusing the source program and for determining the context of the current line. The commands are as follows:

p   prints the current line

w	window; prints a window of ten lines around the current line
z	prints ten lines starting at the current line; advances the current line by ten
control-d	scrolls; prints the next ten lines and advances the current line by ten; used to cleanly display long segments of the program

When a line from a file is printed, it is preceded by its line number. This not only gives an indication of its relative position in the file, but it is also used as input by some **sdb** commands.

Changing the Current Source File or Function

The **e** command is used to change the current source file. Either of the forms

```
*e function
*e file.c
```

may be used. The first causes the file containing the named function to become the current file, and the current line becomes the first line of the function. The other form causes the named file to become current. In this case, the current line is set to the first line of the named file. Finally, an **e** command with no argument causes the current function and file named to be printed.

Changing the Current Line in the Source File

The **z** and **control-d** commands have a side effect of changing the current line in the source file. The following paragraphs describe other commands that change the current line.

There are two commands for searching for instances of regular expressions in source files. They are

```
*/regular expression/
*?regular expression?
```

The first command searches forward through the file for a line containing a string that matches the regular expression, and the second searches backwards. The trailing / and ? may be omitted from these commands. Regular expression matching is identical to that of **ed**(1).

The + and − commands may be used to move the current line forward or backward by a specified number of lines. Typing a new-line advances the current line by one, and typing a number causes that line to become the current line in the file. These commands may be combined with the display commands so that

       *+15z

advances the current line by 15 and then prints ten lines.

A Controlled Environment for Program Testing

One very useful feature of **sdb** is breakpoint debugging. After entering **sdb**, breakpoints can be set at certain lines in the source program. The program is then started with an **sdb** command. Execution of the program proceeds as normal until it is about to execute one of the lines at which a breakpoint has been set. The program stops and **sdb** reports the breakpoint where the program stopped. Now, **sdb** commands may be used to display the trace of function calls and the values of variables. If the user is satisfied the program is working correctly to this point, some breakpoints can be deleted and others set; then program execution may be continued from the point where it stopped.

A useful alternative to setting breakpoints is single-stepping. **sdb** can be requested to execute the next line of the program and then stop. This feature is especially useful for testing new programs, so they can be verified on a statement-by-statement basis. If an attempt is made to single-step through a function that has not been compiled with the −g option, execution proceeds until a statement in a function compiled with the −g option is reached. It is also possible to have the program execute one machine level instruction at a time. This is particularly useful when the program has not been compiled with the −g option.

Setting and Deleting Breakpoints

Breakpoints can be set at any line in a function compiled with the −g option. The command format is as follows:

       *12b
       *proc:12b
       *proc:b
       *b

The first form sets a breakpoint at line 12 in the current file. The line

numbers are relative to the beginning of the file as printed by the source file display commands. The second form sets a breakpoint at line 12 of function **proc**, and the third sets a breakpoint at the first line of **proc**. The last sets a breakpoint at the current line.

Breakpoints are deleted similarly with the **d** command:

```
*12d
*proc:12d
*proc:d
```

In addition, if the command **d** is given alone, the breakpoints are deleted interactively. Each breakpoint location is printed, and a line is read from the user. If the line begins with a **y** or **d**, the breakpoint is deleted.

A list of the current breakpoints is printed in response to a **B** command, and the **D** command deletes all breakpoints. It is sometimes desirable to have **sdb** automatically perform a sequence of commands at a breakpoint and then have execution continue. This is achieved with another form of the **b** command.

```
*12b t;x/
```

causes both a trace back and the value of x to be printed each time execution gets to line 12. The **a** command is a variation of the above command. There are two forms:

```
*proc:a
*proc:12a
```

The first prints the function name and its arguments each time it is called, and the second prints the source line each time it is about to be executed. For both forms of the **a** command, execution continues after the function name or source line is printed.

Running the Program

The **r** command is used to begin program execution. It restarts the program as if it were invoked from the shell. The command

```
*r args
```

runs the program with the given arguments as if they had been typed on the shell command line. If no arguments are specified, then the arguments from the last execution of the program within **sdb** are used. To run a program with no arguments, use the **R** command.

After the program is started, execution continues until a breakpoint is encountered, a signal such as INTERRUPT or QUIT occurs, or the program terminates. In all cases after an appropriate message is printed, control returns to the user.

The **c** command may be used to continue execution of a stopped program. A line number may be specified, as in

*proc: 12c

This places a temporary breakpoint at the named line. The breakpoint is deleted when the **c** command finishes. There is also a **C** command that continues but passes the signal that stopped the program back to the program. This is useful for testing user-written signal handlers. Execution may be continued at a specified line with the **g** command. For example,

*17 g

continues at line 17 of the current function. A use for this command is to avoid executing a section of code that is known to be bad. The user should not attempt to continue execution in a function different from that of the breakpoint.

The **s** command is used to run the program for a single statement. It is useful for slowly executing the program to examine its behavior in detail. An important alternative is the **S** command. This command is like the **s** command but does not stop within called functions. It is often used when one is confident that the called function works correctly but is interested in testing the calling routine.

The **i** command is used to run the program one machine-level instruction at a time, while ignoring the signal that stopped the program. Its uses are similar to the **s** command. There is also an **I** command that causes the program to execute one machine-level instruction at a time, but also passes the signal that stopped the program back to the program.

Calling Functions

It is possible to call any of the functions of the program from **sdb**. This feature is useful both for testing individual functions with different arguments and for calling a user-supplied function to print-structured data. There are two ways to call a function:

*proc(arg1, arg2, ...)
*proc(arg1, arg2, ...)/*m*

The first simply executes the function. The second is intended for calling functions (it executes the function and prints the value that it returns). The value is printed in decimal unless some other format is specified by *m*. Arguments to functions may be integer, character or string constants, or variables that are accessible from the current function.

An unfortunate bug in the current implementation is that if a function is called when the program is not stopped at a breakpoint (such as when a core image is being debugged) all variables are initialized before the function is started. This makes it impossible to use a function that formats data from a dump.

Machine Language Debugging

The **sdb** program has facilities for examining programs at the machine language level. It is possible to print the machine language statements associated with a line in the source and to place breakpoints at arbitrary addresses. The **sdb** program can also be used to display or modify the contents of the machine registers.

Displaying Machine Language Statements

To display the machine language statements associated with line 25 in function **main**, use the command

    *main:25?

The **?** command is identical to the **/** command except that it displays from text space. The default format for printing text space is the **i** format, which interprets the machine language instruction. The **control-d** command may be used to print the next ten instructions.

Absolute addresses may be specified instead of line numbers by appending a **:** to them so that

    *0x1024:?

displays the contents of address 0x1024 in text space. Note that the command

    *0x1024?

displays the instruction corresponding to line 0x1024 in the current function. It is also possible to set or delete a breakpoint by specifying its absolute address;

```
*0x1024:b
```

sets a breakpoint at address 0x1024.

Manipulating Registers

The **x** command prints the values of all the registers. Also, individual registers may be named by appending a % sign to their name so that on the 80286

```
*ax%
```

displays the value of register **ax**, and on the 80386

```
*eax%
```

displays the value of register **eax**.

Other Commands

To exit **sdb**, use the **q** command.

The **!** command (when used immediately after the * prompt) is identical to that in **ed**(1) and is used to have the shell execute a command. The **!** can also be used to change the values of variables or registers when the program is stopped at a breakpoint.

```
*variable!value
*eax!value
```

which sets the variable or the named register to the given value. The value may be a number, character constant, register, or the name of another variable. If the variable is of type **float** or **double**, the value can also be a floating-point constant (specified according to the standard C language format).

An sdb Session

An example of a debugging session using **sdb** is shown in Figure 15-1. Comments (preceded by a pound sign, #) have been added to help you see what is happening.

```
sdb myoptim - .:../common  # enter sdb command
Source path: .:../common
No core image
*window:b                    # set a breakpoint at start of window
0x80802462 (window:1459+2) b
*r < m.s > out.m.s           # run the program
Breakpoint at
0x80802462 in window:1459: window(size, func) register int size;
boolean(*func)(); {
*t                           # print stack trace
window(size=2,func=w2opt)    [optim.c:1459]
peep()  [peep.c:34]
pseudo(s=.def^Imain;^I.val^I.;^I.scl^I-1;^I.endef)  [local.c:483]
yylex()  [local.c:229]
main(argc=0,argv=0xc00201bc,-1073610300)  [optim.c:227]
```

Figure 15-1: Example of **sdb** Usage (Sheet 1 of 3)

```
*z                       # print 10 lines of source
1459: window(size, func) register int size; boolean (*func)(); {
1460:
1461:   extern NODE *initw();
1462:   register NODE *pl;
1463:   register int i;
1464:
1465:   TRACE(window);
1466:
1467:   /* find first window */
1468:
*s                       # step
window: 1459: window(size, func) register int size; boolean (*func)(); {
*s                       # step
window: 1465:    TRACE(window);
*s                       # step
window: 1469:    wsize = size;
*s                       # step
window: 1470:    if ((pl = initw(n0.forw)) == NULL)
*S                       # step through procedure call
window: 1475:    for (opf = pf->back; ; opf = pf->back) {
*pl                      # show variable pl
0x80886b38
*x                       # print the 80286 register contents
   es/ 0x5f            dx/ 0x67            ds/ 0x5f
   ax/ 0x1cea          cd/ 0x57            ip/ 0x40
   di/ 0xcfbd4         cs/ 0x57            bx/ 0xce6
 flgs/ 0x202           bp/ 0x1cd6          sp/ 0x1cce
   si/ 0x5b50          ss/ 0x5f
0x570040 (main+4): lea -8(%bp),%d: [0x7fffffff]
*x                       # print the 80386 register contents
  eax/ 1              ecx/ 0x402e2c       edx/ 0xbffffca8
  eox/ 0x17           esp/ 0xbffffc54     edp/ 0xbffffc60
  esi/ 0x16           edi/ 0x15           eip/ 0x16b
 flag/ 0x13296        trap/ 0xe           err/ 7
0x16b (main+4):    lea   -8(%ebp),%edi   [0x7fffffff]
```

Figure 15-1: Example of **sdb** Usage (Sheet 2 of 3)

```
*pl[0]                    # dereference the pointer
pl[0].forw/ 0x80886b6c
pl[0].back/ 0x80886ac8
pl[0].ops[0]/ pushw
pl[0].uniqid/ 0
pl[0].op/ 123
pl[0].nlive/ 3588
pl[0].ndead/ 4096
*pl->forw[0]              # dereference the pointer
pl->forw[0].forw/ 0x80886ca0
pl->forw[0].back/ 0x80886b38
pl->forw[0].ops[0]/ call
pl->forw[0].uniqid/ 0
pl->forw[0].op/ 9
pl->forw[0].nlive/ 3584
pl->forw[0].ndead/ 4099
*pl!pl->forw              # replace pl with pl->forw
*pl                       # show pl
0x80886b6c
*c                        # continue
Breakpoint at
0x80802462 in window: 1459: window(size, func) register int size;
boolean (*func)(); {
*s                        # step
window: 1459: window(size, func) register int size; boolean (*func)(); {
*s                        # step
window: 1465:   TRACE(window);
*size                     # show function argument size
3
*D                        # delete all breakpoints
All breakpoints deleted
*c                        # continue
Process terminated
*q                        # quit sdb
$
```

Figure 15-1: Example of **sdb** Usage (Sheet 3 of 3)

16 lint

Introduction

The **lint** program examines C language source programs detecting a number of bugs and obscurities. It enforces the type rules of C language more strictly than the C compiler. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful or error-prone constructions, which nevertheless are legal. **lint** accepts multiple input files and library specifications and checks them for consistency.

Usage

The **lint** command has the form

    lint [*options*] *files ... library-descriptors ...*

where *options* are optional flags to control **lint** checking and messages; *files* are the files to be checked which end with **.c** or **.ln**; and *library-descriptors* are the names of libraries to be used in checking the program.

The options that are currently supported by the **lint** command are the following:

-a	suppresses messages about assignments of long values to variables that are not long
-b	suppresses messages about break statements that cannot be reached
-c	checks only for intra-file bugs; leaves external information in files suffixed with **.ln**
-h	does not apply heuristics (which attempt to detect bugs, improve style, and reduce waste)
-n	does not check for compatibility with either the standard or the portable **lint** library
-o *name*	creates a lint library from input files named **llib-l**name**.ln**
-p	checks portability
-u	suppresses messages about function and external variables used and not defined or defined and not used
-v	suppresses messages about unused arguments in functions
-x	does not report variables referred to by external declarations but never used

When more than one option is used, they should be combined into a single argument, such as **-ab** or **-xha**.

The names of files that contain C language programs should end with the suffix **.c**, which is mandatory for **lint** and the C compiler.

The **lint** command accepts certain arguments, such as

-lm

These arguments specify libraries that contain functions used in the C language program. The source code is tested for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library arguments. These files all begin with the comment

`/* LINTLIBRARY */`

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED comments can be used to specify features of the library functions. The next section, **"lint** Message Types,**"** describes how it is done.

lint library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined in a library file but are not used in a source file do not result in messages. **lint** does not simulate a full library search algorithm and will print messages if the source files contain a redefinition of a library routine.

By default, **lint** checks the programs it is given against a standard library file that contains descriptions of the programs that are normally loaded when a C language program is run. When the **-p** option is used, another file is checked containing descriptions of the standard library routines which are expected to be portable across various machines. The **-n** option can be used to suppress all library checking.

lint **Message Types**

The following paragraphs describe the major categories of messages printed by **lint**.

Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused. It is not uncommon for external variables or even entire functions to become unnecessary and yet not be removed from the source. These types of errors rarely cause working programs to fail, but are a source of inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions can occasionally serve to discover bugs.

lint prints messages about variables and functions which are defined but not otherwise mentioned, unless the message is suppressed by means of the **–u** or **–x** option.

Certain styles of programming may permit a function to be written with an interface where some of the function's arguments are optional. Such a function can be designed to accomplish a variety of tasks depending on which arguments are used. Normally **lint** prints messages about unused arguments; however, the **–v** option is available to suppress the printing of these messages. When **–v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This can be considered an active (and preventable) waste of the register resources of the machine.

Messages about unused arguments can be suppressed for one function by adding the comment

        /* ARGSUSED */

to the source code before the function. This has the effect of the **–v** option for only one function. Also, the comment

        /* VARARGS */

can be used to suppress messages about a variable number of arguments in calls to a function. The comment should be added before the function definition. In some cases, it is desirable to check the first several arguments and leave the later arguments unchecked. This can be done with a digit giving the number of arguments which should be checked. For example,

```
/* VARARGS2 */
```

will cause only the first two arguments to be checked.

When **lint** is applied to some, but not all, files out of a collection that are to be loaded together, it issues complaints about unused or undefined variables. This information is, of course, more distracting than helpful. Functions and variables that are defined may not be used; conversely, functions and variables defined elsewhere may be used. The **-u** option suppresses the spurious messages.

Set/Used Information

lint attempts to detect cases where a variable is used before it is set. It detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a "use," since the actual use may occur at any later time, in a data-dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does mean that **lint** can print error messages about program fragments that are legal, but these programs would probably be considered bad on stylistic grounds. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. The **lint** program does deal with initialized automatic variables.

The set/used information also permits recognition of those local variables that are set and never used. These form a frequent source of inefficiencies and may also be symptomatic of bugs.

Flow of Control

lint attempts to detect unreachable portions of a program. It will print messages about unlabeled statements immediately following **goto, break, continue**, or **return** statements. It attempts to detect loops that cannot be left at the bottom and to recognize the special cases **while(1)** and **for(;;)** as infinite loops. **lint** also prints messages about loops that cannot be entered at the top. Valid programs may have such loops, but they are considered to be bad style. If you do not want messages about unreached portions of the program, use the **-b** option.

lint has no way of detecting functions that are called and never return. Thus, a call to **exit** may cause unreachable code which **lint** does not detect. The most serious effects of this are in the determination of returned function values (see "Function Values"). If a particular place in the program is thought to be unreachable in a way that is not apparent to **lint**, the comment

```
/* NOTREACHED */
```

can be added to the source code at the appropriate place. This comment will inform **lint** that a portion of the program cannot be reached, and **lint** will not print a message about the unreachable portion.

Programs generated by **yacc** and especially **lex** may have hundreds of unreachable **break** statements, but messages about them are of little importance. There is typically nothing the user can do about them, and the resulting messages would clutter up the **lint** output. The recommendation is to invoke **lint** with the −**b** option when dealing with such input.

Function Values

Sometimes functions return values that are never used. Sometimes programs incorrectly use function values that have never been returned. **lint** addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return( expr );
```

and

```
return ;
```

statements is cause for alarm; **lint** will give the message

```
function name has return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {
        if ( a ) return ( 3 );
        g ();
        }
```

Notice that, if **a** tests false, **f** will call **g** and then return with no defined return value; this will trigger a message from **lint**. If **g**, like **exit**, never returns, the message will still be produced, when in fact nothing is wrong. A comment

```
/*NOTREACHED*/
```

in the source code will cause the message to be suppressed. In practice, some potentially serious bugs have been discovered by this feature.

On a global scale, **lint** detects cases where a function returns a value that is sometimes or never used. When the value is never used, it may constitute an inefficiency in the function definition that can be overcome by specifying the function as being of type (void). For example:

```
(void) fprintf(stderr,"File busy. Try again later!\n");
```

When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The opposite problem, using a function value when the function does not return one, is also detected. This is a serious problem.

Type Checking

lint enforces the type checking rules of C language more strictly than the compilers do. The additional checking is in four major areas:

- across certain binary operators and implied assignments

- at the structure selection operators

- between the definition and uses of functions

- in the use of enumerations

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (**?:**), and relational operators have this property. The argument of a **return** statement and expressions used in initialization suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly except that arrays of xs can, of course, be intermixed with pointers to xs.

The type checking rules also require that, in structure references, the left operand of the -> be a pointer to structure, the left operand of the . be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types or other enumerations and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

If it is desired to turn off strict type checking for an expression, the comment

```
/* NOSTRICT */
```

should be added to the source code immediately before the expression. This comment will prevent strict type checking for only the next line in the program.

Type Casts

The type cast feature in C language was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where **p** is a character pointer. **lint** will print a message as a result of detecting this. Consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this and has clearly signaled his intentions. Nevertheless, **lint** will continue to print messages about this.

Nonportable Character Use

On some systems, characters are signed quantities with a range from −128 to 127. On other C language implementations, characters take on only positive values. Thus, **lint** will print messages about certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
    ...
if( (c = getchar( )) < 0 ) ...
```

will work on one machine but will fail on machines where characters always take on positive values. The real solution is to declare c as an integer since **getchar** is actually returning integer values. In any case, **lint** will print the message

```
nonportable character comparison
```

A similar issue arises with bit fields. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is especially true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a two-bit field declared of type **int** cannot hold the value 3, the problem disappears if the bit field is declared to have type **unsigned**.

Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which will truncate the contents. This may happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, which are truncated. The −a option can be used to suppress messages about the assignment of **longs** to **ints**.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are detected by **lint**. It is hoped that the messages encourage better code quality, clearer style, and even point out bugs. The **-h** option is used to suppress these checks. For example, in the statement

```
*p++ ;
```

the * does nothing. This provokes the message

```
null effect
```

from **lint**. The following program fragment

```
unsigned x ;
if( x < 0 ) ...
```

results in a test that will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. **lint** will print the message

```
degenerate unsigned comparison
```

in these cases. If a program contains something similar to

```
if( 1 != 0 ) ...
```

lint will print the message

```
constant in conditional context
```

since the comparison of 1 with 0 gives a constant result.

Another construction detected by **lint** involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if( x&077 == 0 ) ...
```

and

```
x<<2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and **lint** encourages this by an appropriate message.

Old Syntax

Several forms of older syntax are now illegal. These fall into two classes: assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =-, ...) could cause ambiguous expressions, such as,

```
a =-1 ;
```

which could be taken as either

```
a =- 1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (e.g., +=, -=, ...) have no such ambiguities. To encourage the abandonment of the older forms, **lint** prints messages about these old-fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int x 1 ;
```

to initialize x to 1. This also caused syntactic difficulties. For example, the initialization

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function definition

```
int x ( y ) { ...
```

and the compiler must read past x in order to determine the correct meaning. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

```
int x = -1 ;
```

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines and illegal on others due entirely to alignment restrictions. **lint** tries to detect cases where pointers are assigned to other pointers and such alignment problems might arise. The message

        possible pointer alignment problem

results from this situation.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine-dependent. For example, on machines in which the stack runs backwards, function arguments will probably be best evaluated from right to left. On machines with a stack running forward, left to right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators that have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C language on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

lint checks for the important special case where a simple scalar variable is affected. For example, the statement

        a[i] = b[i++];

will cause **lint** to print the message

        warning: i evaluation order undefined

in order to call attention to this condition.

17 C Language

Constant Expressions

Portability Considerations

Syntax Summary

Introduction

This chapter contains a summary of the grammar and syntax rules of the C Programming Language. A consistent attempt is made to point out where other implementations may differ.

Lexical Conventions

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Blanks, tabs, new-lines, and comments (collectively, "white space") as described below are ignored, except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token.

Comments

The characters /* introduce a comment that terminates with the characters */. Comments do not nest.

Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore (_) counts as a letter. Uppercase and lowercase letters are different. There is no limit on the length of a name. Other implementations may collapse case distinctions for external names, and may reduce the number of significant characters for both external and non-external names.

Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

asm	default	float	register	switch
auto	do	for	return	typedef
break	double	goto	short	union
case	else	if	sizeof	unsigned
char	enum	int	static	void
continue	external	long	struct	while

Some implementations also reserve the word **fortran.**

Constants

There are several kinds of constants. Each has a type; an introduction to types is given in "Storage Class and Type."

Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with 0 (digit zero). An octal constant consists of the digits 0 through 7 only. A sequence of digits preceded by 0x or 0X (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include a or A through f or F with values 10 through 15. Otherwise, the integer constant is taken to be decimal. A decimal constant whose value exceeds the largest signed machine integer is taken to be **long**; an octal or hex constant that exceeds the largest unsigned machine integer is likewise taken to be **long**. Otherwise, integer constants are **int**.

Explicit Long Constants

A decimal, octal, or hexadecimal integer constant immediately followed by l (letter ell) or L is a long constant. As discussed below, integer and long values may be considered identical.

Character Constants

A character constant is a character enclosed in single quotes, as in 'x'. The value of a character constant is the numerical value of the character in the machine's character set. Certain nongraphic characters, the single quote (') and the backslash (\), may be represented according to the table of escape sequences shown in Figure 17-1:

```
new-line NL (LF)   \n
horizontal tab     HT      \t
vertical tab       VT      \v
backspaceBS        \b
carriage return    CR      \r
form feedFF        \f
backslash\         \\
single quote       '       \'
bit pattern        ddd     \ddd
```

Figure 17-1: Escape Sequences for Nongraphic Characters

━━

The escape \ddd consists of the backslash followed by 1, 2, or 3 octal digits that are taken to specify the value of the desired character. A special case of this construction is \0 (not followed by a digit), which indicates the ASCII character **NUL**. If the character following a backslash is not one of those specified, the behavior is undefined. An explicit new-line character is illegal in a character constant. The type of a character constant is **int**.

Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an **e** or **E**, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing. Either the decimal point or the **e** and the exponent (not both) may be missing. Every floating constant has type **double**.

Enumeration Constants

Names declared as enumerators (see "Structure and Union Declarations" and "Enumeration Declarations" under "Declarations") have type **int**.

String Literals

A string literal is a sequence of characters surrounded by double quotes, as in "...". A string literal has type "array of **char**" and storage class **static** (see "Storage Class and Type") and is initialized with the given characters. The compiler places a null byte (\0) at the end of each string literal so that programs that scan the string literal can find its end. In a string literal, the double quote character (") must be preceded by a \; in addition, the same escapes as described for character constants may be used.

A \ and the immediately following new-line are ignored. All string literals, even when written identically, are distinct.

Syntax Notation

Syntactic categories are indicated by *italic* type and literal words and characters by **bold** type. Alternative categories are listed on separate lines. An optional entry is indicated by the subscript "opt," so that

   { *expression $_{opt}$* }

indicates an optional expression enclosed in braces. The syntax is summarized in "Syntax Summary" at the end of the chapter.

Storage Class and Type

The C language bases the interpretation of an identifier upon two attributes of the identifier: its storage class and its type. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

Storage Class

There are four declarable storage classes:

- automatic
- static
- external
- register

Automatic variables are local to each invocation of a block (see "Compound Statement or Block" in "Statements") and are discarded upon exit from the block. Static variables are local to a block but retain their values upon reentry to a block even after control has left the block. External variables exist and retain their values throughout the execution of the entire program and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables, they are local to each block and disappear on exit from the block.

Type

The C language supports several fundamental types of objects. Objects declared as characters (**char**) are large enough to store any member of the implementation's character set. If a genuine character from that character set is stored in a **char** variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine-dependent. In particular, **char** may be signed or unsigned by default. In this implementation the default is signed.

Up to three sizes of integer, declared **short int**, **int**, and **long int**, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. Plain integers have the natural size suggested by the host machine architecture. The other sizes are provided to meet special needs. The sizes for the Intel 80286 and Intel 80386 computers are shown in Figure 17-2.

ASCII	80286 COMPUTER bits	80386 COMPUTER bits
char	8	8
int	16	32
short	16	16
long	32	32
float	32	32
double	64	64
float range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 308}$	$\pm 10^{\pm 308}$

Figure 17-2: Computer Hardware Characteristics

The properties of **enum** types (see "Structure and Union Declarations" and "Enumeration Declarations" under "Declarations") are identical to those of some integer types. The implementation may use the range of values to determine how to allot storage.

Unsigned integers, declared **unsigned,** obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation.

Single-precision floating point (**float**) and double precision floating point (**double**) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as arithmetic types. **Char, int** of all sizes whether **unsigned** or not, and **enum** will collectively be called integral types. The **float** and **double** types will collectively be called floating types.

The **void** type specifies an empty set of values. It is used as the type returned by functions that generate no value.

Besides the fundamental arithmetic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- arrays of objects of most types

- functions that return objects of a given type

- pointers to objects of a given type

- structures containing a sequence of objects of various types

- unions capable of containing any one of several objects of various types

In general these methods of constructing objects can be applied recursively.

Objects and lvalues

An object is a manipulatable region of storage. An lvalue is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators that yield lvalues: for example, if E is an expression of pointer type, then *E is an lvalue expression referring to the object to which E points. The name "lvalue" comes from the assignment expression E1 = E2 in which the left operand E1 must be an lvalue expression. In the following text the discussion of each operator indicates whether it expects lvalue operands and whether it yields an lvalue.

Operator Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This part explains the result to be expected from such conversions. The conversions demanded by most ordinary operators are summarized under "Arithmetic Conversions." The summary will be supplemented as required by the discussion of each operator.

Characters and Integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer preserves sign. On your computer, sign extension of **char** variables does occur. It is guaranteed that a member of the standard character set is non-negative.

On machines that treat characters as signed, the characters of the ASCII set are all non-negative. However, a character constant specified with an octal escape suffers sign extension and may appear negative; for example, '\377' has the value **–1**.

When a longer integer is converted to a shorter integer or to a **char**, it is truncated on the left. Excess bits are simply discarded.

Float and Double

All floating arithmetic in C is carried out in double precision. Whenever a **float** appears in an expression, it is lengthened to **double** by zero padding its fraction. When a **double** must be converted to **float**, for example by an assignment, the **double** is rounded before truncation to **float** length. This result is undefined if it cannot be represented as a float.

Floating and Integral

Conversions of floating values to integral type are rather machine-dependent. In particular, the direction of truncation of negative numbers varies. The result is undefined if it will not fit in the space provided.

Conversions of integral values to floating type behave well. Some loss of accuracy occurs if the destination lacks sufficient bits.

Pointers and Integers

An expression of integral type may be added to or subtracted from a pointer; in such a case, the first is converted as specified in the discussion of the addition operator. Two pointers to objects of the same type may be subtracted; in this case, the result is converted to an integer as specified in the discussion of the subtraction operator.

Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo $2^{wordsize}$). In a 2's complement representation, this conversion is conceptual; and there is no actual change in the bit pattern.

When an unsigned **short** integer is converted to **long,** the value of the result is the same numerically as that of the unsigned integer. Thus, the conversion amounts to padding with zeros on the left.

Arithmetic Conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

1. First, any operands of type **char** or **short** are converted to **int**, and any operands of type **unsigned char** or **unsigned short** are converted to **unsigned int**.

2. Then, if either operand is **double,** the other is converted to **double** and that is the type of the result.

3. Otherwise, if either operand is **unsigned long**, the other is converted to **unsigned long** and that is the type of the result.

4. Otherwise, if one operand is **long**, and the other is **unsigned int**, they are both converted to **unsigned long** and that is the type of the result.

5. Otherwise, if either operand is **long**, the other is converted to **long** and that is the type of the result.

6. Otherwise, if either operand is **unsigned,** the other is converted to **unsigned** and that is the type of the result.

7. Otherwise, both operands must be **int**, and that is the type of the result.

Void

The (nonexistent) value of a **void** object may not be used in any way, and neither explicit nor implicit conversion may be applied. Because a void expression denotes a nonexistent value, such an expression may be used only as an expression statement (see "Expression Statement" under "Statements") or as the left operand of a comma expression (see "Comma Operator" under "Expressions and Operators").

An expression may be converted to type **void** by use of a cast. For example, this makes explicit the discarding of the value of a function call used as an expression statement.

Expressions and Operators

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (see "Additive Operators") are those expressions defined under "Primary Expressions", "Unary Operators", and "Multiplicative Operators". Within each subpart, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators are summarized in the grammar of "Syntax Summary".

Otherwise, the order of evaluation of expressions is undefined. In particular, the compiler considers itself free to compute subexpressions in the order it believes most efficient even if the subexpressions involve side effects. Expressions involving a commutative and associative operator (*, +, &, !, ^) may be rearranged arbitrarily even in the presence of parentheses; to force a particular order of evaluation, an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is undefined. Most existing implementations of C ignore integer overflows; treatment of division by 0 and all floating-point exceptions varies between machines and is usually adjustable by a library function.

Primary Expressions

Primary expressions involving ., ->, subscripting, and function calls group left to right.

> *primary-expression:*
>   *identifier*
>   *constant*
>   *string literal*
>   *(expression)*
>   *primary-expression [expression]*
>   *primary-expression (expression-list_{opt})*
>   *primary-expression . identifier*
>   *primary-expression -> identifier*

expression-list:
    expression
    expression-list , expression

An identifier is a primary expression provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of . . .", then the value of the identifier expression is a pointer to the first object in the array; and the type of the expression is "pointer to . . .". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier that is declared "function returning . . .", when used except in the function-name position of a call, is converted to "pointer to function returning . . .".

A constant is a primary expression. Its type may be **int**, **long**, or **double** depending on its form. Character constants have type **int**, and floating constants have type **double**.

A string literal is a primary expression. Its type is originally "array of **char**", but following the same rule given above for identifiers, this is modified to "pointer to **char**" and the result is a pointer to the first character in the string literal. (There is an exception in certain initializers; see "Initialization" under "Declarations.")

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to . . .", the subscript expression is **int**, and the type of the result is " . . . ". The expression **E1[E2]** is identical (by definition) to ***((E1)+(E2))**. All the clues needed to understand this notation are contained in this subpart together with the discussions in "Unary Operators" and "Additive Operators" on identifiers, ***** and **+,** respectively. The implications are summarized under "Arrays, Pointers, and Subscripting" under "Types Revisited."

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions that constitute the actual arguments to the function. The primary expression must be of type "function returning ...", and the result of the function call is of type " ... ". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer.

Any actual arguments of type **float** are converted to **double** before the call. Any of type **char** or **short** are converted to **int**. Array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see "Unary Operators" and "Type Names" under "Declarations."

In preparing for the call to a function, a copy is made of each actual parameter. Thus, all argument passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. It is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ. Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from – and >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union, and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points. Thus the expression **E1->MOS** is the same as **(*E1).MOS**. Structures and unions are discussed in "Structure and Union Declarations" and "Enumeration Declarations" under "Declarations."

Unary Operators

Expressions with unary operators group right to left.

> *unary-expression:*
>   * *expression*
>   & *lvalue*
>   – *expression*
>   ! *expression*
>   ~ *expression*
>   ++ *lvalue*
>   —*lvalue*
>   *lvalue* ++
>   *lvalue* --
>   (*type-name*) *expression*
>   sizeof *expression*
>   sizeof (*type-name*)

The unary * operator means "indirection"; the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...," the type of the result is " ... ".

The result of the unary & operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is " ... ", the type of the result is "pointer to ...".

The result of the unary – operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n, where n is the number of bits in the corresponding signed type.

There is no unary + operator.

The result of the logical negation operator ! is one if the value of its operand is zero, and zero if the value of its operand is nonzero. The type of the result is **int**. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand but is not an lvalue. The expression ++**x** is equivalent to **x** += **1**. See the discussions "Additive Operators" and "Assignment Operators" for information on conversions.

The lvalue operand of prefix -- is decremented analogously to the prefix ++ operator.

When postfix ++ is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix -- is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix -- operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a cast. Type names are described in "Type Names" under "Declarations."

The **sizeof** operator yields the size in bytes of its operand. (A byte is undefined by the language except in terms of the value of **sizeof**. However, in all existing implementations, a byte is the space required to hold a **char**.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an **unsigned** constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The **sizeof** operator may also be applied to a parenthesized type name. In that case it yields the size in bytes of an object of the indicated type.

The construction **sizeof(**type**)** is taken to be a unit, so the expression **sizeof(**type**)–2** is the same as **(sizeof(**type**))–2**.

Multiplicative Operators

The multiplicative operators *****, **/**, and **%** group left to right. The usual arithmetic conversions are performed.

> *multiplicative expression:*
> > *expression* * *expression*
> > *expression* / *expression*
> > *expression* % *expression*

The binary * operator indicates multiplication. The * operator is associative, and expressions with several multiplications at the same level may be rearranged by the compiler. The binary / operator indicates division.

The binary % operator yields the remainder from the division of the first expression by the second. The operands must be integral.

When positive integers are divided, truncation is toward 0; but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that **(a/b)*b** + **a%b** is equal to **a** (if **b** is not 0).

Additive Operators

The additive operators + and − group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

> *additive-expression:*
> > *expression* + *expression*
> > *expression* − *expression*

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer that points to another object in the same array, appropriately offset from the original object. Thus if **P** is a pointer to an object in an array, the expression **P+1** is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The + operator is associative, and expressions with several additions at the same level may be rearranged by the compiler.

The result of the – operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an **unsigned** representing the difference of the indices of the pointed-to objects in their array. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

Shift Operators

The shift operators << and >> group left to right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to **int**; the type of the result is that of the left operand. The result is undefined if the right operand is negative or greater than or equal to the length of the object in bits.

> *shift-expression:*
> > *expression* << *expression*
> > *expression* >> *expression*

The value of **E1<<E2** is **E1** (interpreted as a bit pattern) left-shifted **E2** bits. Vacated bits are 0 filled. The value of **E1>>E2** is **E1** right-shifted **E2** bit positions. The right shift is guaranteed to be logical (0 fill) if **E1** is **unsigned**; otherwise, it may be arithmetic.

Relational Operators

The relational operators group left to right.

> *relational-expression:*
> > *expression* < *expression*
> > *expression* > *expression*
> > *expression* <= *expression*
> > *expression* >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is **int**. The usual arithmetic conversions are

performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

Equality Operators

> *equality-expression:*
>   *expression* == *expression*
>   *expression* != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus **a<b == c<d** is 1 whenever **a<b** and **c<d** have the same truth value.)

A pointer may be compared to an integer only if the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object and will appear to be equal to 0. In conventional usage, such a pointer is considered to be null.

Bitwise AND Operator

> *and-expression:*
>   *expression* & *expression*

The **&** operator is associative, and expressions involving **&** may be rearranged. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands. The operator applies only to integral operands.

Bitwise Exclusive OR Operator

> *exclusive-or-expression:*
>   *expression* ^ *expression*

The ^ operator is associative, and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

Bitwise Inclusive OR Operator

inclusive-or-expression:
    expression I expression

The I operator is associative, and expressions involving I may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

Logical AND Operator

logical-and-expression:
    expression && expression

The **&&** operator groups left to right. It returns 1 if both its operands evaluate to nonzero, 0 otherwise. Unlike **&**, **&&** guarantees left to right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

Logical OR Operator

logical-or-expression:
    expression ‖ expression

The ‖ operator groups left to right. It returns 1 if either of its operands evaluates to nonzero, 0 otherwise. Unlike ¦, ‖ guarantees left to right evaluation; moreover, the second operand is not evaluated if the value of the first operand evaluates to nonzero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always **int**.

Conditional Operator

conditional-expression:
> *expression ? expression : expression*

Conditional expressions group right to left. The first expression is evaluated; and if it is nonzero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type. If both are structures or unions of the same type, the result has the type of the structure or union. If both pointers are of the same type, the result has the common type. Otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

Assignment Operators

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
> *lvalue = expression*
> *lvalue += expression*
> *lvalue -= expression*
> *lvalue *= expression*
> *lvalue /= expression*
> *lvalue %= expression*
> *lvalue >>= expression*
> *lvalue <<= expression*
> *lvalue &= expression*
> *lvalue ^= expression*
> *lvalue != expression*

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. Second, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand must in general be a pointer of the same type. However, the constant 0 may be assigned to a pointer; it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

The behavior of an expression of the form **E1** *op* = **E2** may be inferred by taking it as equivalent to **E1** = **E1** *op* (**E2**); however, **E1** is evaluated only once. In += and −=, the left operand may be a pointer, in which case the (integral) right operand is converted as explained in "Additive Operators." All right operands and all nonpointer left operands must have arithmetic type.

Comma Operator

>*comma-expression:*
>>*expression , expression*

A pair of expressions separated by a comma is evaluated left to right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left to right. In contexts where comma is given a special meaning, e.g., in lists of actual arguments to functions (see "Primary Expressions") and lists of initializers (see "Initialization" under "Declarations"), the comma operator as described in this subpart can only appear in parentheses. For example,

>f(a, (t=3, t+2), c)

has three arguments, the second of which has the value 5.

Declarations

Declarations are used to specify the interpretation that C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

> *declaration:*
> > *decl-specifiers declarator-list$_{opt}$;*

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

> *decl-specifiers:*
> > *type-specifier decl-specifiers$_{opt}$*
> > *sc-specifier decl-specifiers$_{opt}$*

The list must be self-consistent in a way described below.

Storage Class Specifiers

The sc-specifiers are:

> *sc-specifier:*
> > **auto**
> > **static**
> > **extern**
> > **register**
> > **typedef**

The **typedef** specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience. See "**typedef**" for more information. The meanings of the various storage classes were discussed in "Storage Class."

The **auto**, **static**, and **register** declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the **extern** case, there must be an external definition (see "External Definitions") for the given identifiers somewhere outside the function in which they are declared.

A **register** declaration is best thought of as an **auto** declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations in each function are effective. Moreover, only variables of certain types will be stored in registers. One other restriction applies to variables declared using register storage class: the address of

operator, &, cannot be applied to them. Smaller, faster programs can be
expected if register declarations are used appropriately.

At most, one sc-specifier may be given in a declaration. If the sc-specifier
is missing from a declaration, it is taken to be **auto** inside a function, **extern**
outside. Exception: functions are never automatic.

Type Specifiers

The type-specifiers are

> *type-specifier:*
>> *struct-or-union-specifier*
>> *typedef-name*
>> *enum-specifier*
> *basic-type-specifier:*
>> *basic-type*
>> *basic-type basic-type-specifiers*
> *basic-type:*
>> **char**
>> **short**
>> **int**
>> **long**
>> **unsigned**
>> **float**
>> **double**
>> **void**

At most, one of the words **long** or **short** may be specified in conjunction with
int; the meaning is the same as if **int** were not mentioned. The word **long**
may be specified in conjunction with **float**; the meaning is the same as **dou-
ble**. The word **unsigned** may be specified alone, or in conjunction with **int** or
any of its short or long varieties, or with **char**.

Otherwise, at most one type-specifier may be given in a declaration. In
particular, adjectival use of **long, short,** or **unsigned** is not permitted with
typedef names. If the type-specifier is missing from a declaration, it is taken
to be **int**.

Specifiers for structures, unions, and enumerations are discussed in "Structure and Union Declarations" and "Enumeration Declarations." Declarations with **typedef** names are discussed in "**typedef.**"

Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer:

> *declarator-list:*
> > *init-declarator*
> > *init-declarator , declarator-list*
>
> *init-declarator:*
> > *declarator initializer$_{opt}$*

Initializers are discussed in "Initialization." The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

> *declarator:*
> > *identifier*
> > *(declarator)*
> > $*$ *declarator*
> > *declarator ()*
> > *declarator [constant-expression$_{opt}$]*

The grouping is the same as in expressions.

Meaning of Declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type.

Each declarator contains exactly one identifier; it is this identifier that is declared. If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

    T D1

where **T** is a type-specifier (like **int**, etc.) and **D1** is a declarator. Suppose this declaration makes the identifier have type " ... **T**", where the " ... " is empty if **D1** is just a plain identifier (so that the type of **x** in "**int x**" is just **int**). Then if **D1** has the form

    *D

the type of the contained identifier is " ... pointer to **T**".

If **D1** has the form

    D()

then the contained identifier has the type " ... function returning **T**."

If **D1** has the form

    D[*constant-expression*]

or

    D[]

then the contained identifier has type " ... array of **T**". In the first case, the constant expression is an expression whose value is determinable at compile time, whose type is **int**, and whose value is positive. (Constant expressions are defined precisely in "Constant Expressions.") When several "array of" specifications are adjacent, a multi-dimensional array is created; the constant expressions that specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multi-dimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays or functions, although they may return pointers; there are no arrays of functions, although there may be arrays of pointers to functions. Likewise, a structure or union may not contain a function; but it may contain a pointer to a function.

As an example, the declaration

    int i, *ip, f(), *fip(), (*pfi)();

declares an integer **i**, a pointer **ip** to an integer, a function **f** returning an integer, a function **fip** returning a pointer to an integer, and a pointer **pfi** to a function, which returns an integer. It is especially useful to compare the last two. The binding of *fip() is *(fip()). The declaration suggests, and the same construction in an expression requires, the calling of a function **fip**, and then using indirection through the (pointer) result to yield an integer. In the declarator (*pfi)(), the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

    float fa[17], *afp[17];

declares an array of **float** numbers and an array of pointers to **float** numbers. Finally,

    static int x3d[3][5][7];

declares a static 3-dimensional array of integers, with rank $3\times5\times7$. In complete detail, **x3d** is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions **x3d, x3d[i], x3d[i][j]**, or **x3d[i][j][k]** may reasonably appear in an expression. The first three have type "array of ..." and the last has type **int**.

Structure and Union Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object that may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

> *struct-or-union-specifier:*
>   *struct-or-union* { *struct-decl-list* }
>   *struct-or-union identifier* { *struct-decl-list* }
>   *struct-or-union identifier*

> *struct-or-union:*
>   **struct**
>   **union**

The struct-decl-list is a sequence of declarations for the members of the structure or union:

> *struct-decl-list:*
>   *struct-declaration*
>   *struct-declaration struct-decl-list*

> *struct-declaration:*
>   *type-specifier struct-declarator-list* ;

> *struct-declarator-list:*
>   *struct-declarator*
>   *struct-declarator , struct-declarator-list*

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a field; its length, a non-negative constant expression, is set off from the field name by a colon.

> *struct-declarator:*
>   *declarator*
>   *declarator : constant-expression*
>   *: constant-expression*

    Within a structure, the objects declared have addresses that increase as the declarations are read left to right. Each non-field member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field that does not fit into the space remaining in a word is put into the next word. No field may be wider than a word. (See Figure 17-2 for sizes of basic types on your 286- and 386-based computer system.)

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, a field with a width of 0 specifies alignment of the next field at an implementation-dependent boundary.

The language does not restrict the types of things that are declared as fields. Moreover, even **int** fields may be considered to be unsigned. For these reasons, it is strongly recommended that fields be declared as **unsigned** where that is the intent. There are no arrays of fields, and the address-of operator, **&**, may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most, one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

> **struct** *identifier* { *struct-decl-list* }
> **union** *identifier* { *struct-decl-list* }

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

> **struct** *identifier*
> **union** *identifier*

Structure tags allow definition of self-referential structures. Structure tags also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union that contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The third form of a structure or union specifier may be used prior to a declaration that gives the complete specification of the structure or union in situations in which the size of the structure or union is unnecessary. The size is unnecessary in two situations: when a pointer to a structure or union is being declared, and when a **typedef** name is declared to be a synonym for a structure or union. This, for example, allows the declaration of a pair of structures that contain pointers to each other.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple but important example of a structure declaration is the following binary tree structure

```
struct tnode
{
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

struct tnode s, *sp;

declares **s** to be a structure of the given sort and **sp** to be a pointer to a structure of the given sort. With these declarations, the expression

sp->count

refers to the **count** field of the structure to which **sp** points;

s.left

refers to the left subtree pointer of the structure **s**; and

s.right->tword[0]

refers to the first character of the **tword** member of the right subtree of **s**.

Enumeration Declarations

Enumeration variables and constants have integral type.

enum-specifier:
    enum { *enum-list* }
    enum *identifier* { *enum-list* }
    enum *identifier*

enum-list:
    enumerator
    enum-list , enumerator

enumerator:
    identifier
    identifier = *constant-expression*

The identifiers in an enum-list are declared as constants and may appear wherever constants are required. If no enumerators with = appear, then the values of the corresponding constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with = gives the associated identifier the value indicated; subsequent identifiers continue the progression from the assigned value.

The names of enumerators in the same scope must all be distinct from each other and from those of ordinary variables.

The role of the identifier in the enum-specifier is entirely analogous to that of the structure tag in a struct-specifier; it names a particular enumeration. For example,

```
enum color { chartreuse, burgundy, claret=20, winedark };
...
enum color *cp, col;
...
col = claret;
cp = &col;
...
if (*cp == burgundy) ...
```

makes **color** the enumeration-tag of a type describing various colors, and then declares **cp** as a pointer to an object of that type and **col** as an object of that type. The possible values are drawn from the set {0,1,20,21}.

Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by = and consists of an expression or a list of values nested in braces.

> *initializer:*
> > = *expression*
> > = { *initializer-list* }
> > = { *initializer-list ,* }
>
> *initializer-list:*
> > *expression*
> > *initializer-list , initializer-list*
> > { *initializer-list* }
> > { *initializer-list ,* }

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in "Constant Expressions," or expressions that reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants and previously declared variables and functions.

Static and external variables that are not initialized are guaranteed to start off as zero. Automatic and register variables that are not initialized are guaranteed to start off as garbage.

When an initializer applies to a scalar (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an aggregate (a structure or array), the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with zeros. It is not permitted to initialize unions or automatic aggregates.

Braces may in some cases be omitted. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate. Any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a **char** array to be initialized by a string literal. In this case successive characters of the string literal initialize the members of the array.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes **x** as a one-dimensional array that has three members, since no size was specified and there are three initializers.

```
float y[4][3] =
{
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array **y[0]**, namely **y[0][0]**, **y[0][1]**, and **y[0][2]**. Likewise, the next two lines initialize **y[1]** and **y[2]**. The initializer ends early and therefore **y[3]** is

initialized with 0. Precisely, the same effect could have been achieved by

    float y[4][3] =
    {
        1, 3, 5, 2, 4, 6, 3, 5, 7
    };

The initializer for **y** begins with a left brace but that for **y[0]** does not; there-
fore, three elements from the list are used. Likewise, the next three are taken
successively for **y[1]** and **y[2]**. Also,

    float y[4][3] =
    {
        { 1 }, { 2 }, { 3 }, { 4 }
    };

initializes the first column of **y** (regarded as a two-dimensional array) and
leaves the rest 0.

    Finally,

    char msg[] = "Syntax error on line %s\n";

shows a character array whose members are initialized with a string literal.
The length of the string (or size of the array) includes the terminating NUL
character, \0.

Type Names

    In two contexts (to specify type conversions explicitly by means of a cast
and as an argument of **sizeof**), it is desired to supply the name of a data type.
This is accomplished using a "type name", which in essence is a declaration
for an object of that type that omits the name of the object.

    type-name:
        type-specifier abstract-declarator

    abstract-declarator:
        empty
        (abstract-declarator)
        ∗ abstract-declarator
        abstract-declarator ()
        abstract-declarator [constant-expression$_{opt}$]

To avoid ambiguity, in the construction

(*abstract-declarator*)

the abstract-declarator is required to be nonempty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
int (*[3])()
```

name respectively the types "integer," "pointer to integer," "array of three pointers to integers," "pointer to an array of three integers," "function returning pointer to integer," "pointer to function returning an integer," and "array of three pointers to functions returning an integer."

Implicit Declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be **int**; if a type but no storage class is indicated, the identifier is assumed to be **auto**. An exception to the latter rule is made for functions because **auto** functions do not exist. If the type of an identifier is "function returning ... ", it is implicitly declared to be **extern**.

In an expression, an identifier followed by (and not already declared is contextually declared to be "function returning **int**".

typedef

Declarations whose "storage class" is **typedef** do not define storage but instead define identifiers that can be used later as if they were type keywords naming fundamental or derived types.

> *typedef-name:*
> *identifier*

Within the scope of a declaration involving **typedef**, each identifier appearing as part of any declarator therein becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in "Meaning of Declarators." For example, after

> **typedef int MILES, *KLICKSP;**
> **typedef struct { double re, im; } complex;**

the constructions

> **MILES distance;**
> **extern KLICKSP metricp;**
> **complex z, *zp;**

are all legal declarations; the type of **distance** is **int**, that of **metricp** is "pointer to **int**", and that of **z** is the specified structure. The **zp** is a pointer to such a structure.

The **typedef** does not introduce brand-new types, only synonyms for types that could be specified in another way. Thus in the example above **distance** is considered to have exactly the same type as any other **int** object.

Statements

Except as indicated, statements are executed in sequence.

Expression Statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

Compound Statement or Block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided:

compound-statement:
    { *declaration-list_{opt} statement-list_{opt}* }

declaration-list:
    declaration
    declaration declaration-list

statement-list:
    statement
    statement statement-list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of **auto** or **register** variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of **static** variables are performed only once when the program begins execution. Inside a block, **extern** declarations do not reserve storage, so initialization is not permitted.

Conditional Statement

The two forms of the conditional statement are

> **if** (*expression*) *statement*
> **if** (*expression*) *statement* **else** *statement*

In both cases, the expression is evaluated; if it is nonzero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. The **else** ambiguity is resolved by connecting an **else** with the last encountered **else**-less **if**.

while **Statement**

The **while** statement has the form:

> **while** (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains nonzero. The test takes place before each execution of the statement.

do **Statement**

The **do** statement has the form:

> **do** *statement* **while** (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes 0. The test takes place after each execution of the statement.

for **Statement**

The **for** statement has the form:

> **for** ($exp-1_{opt}$; $exp-2_{opt}$; $exp-3_{opt}$) *statement*

Except for the behavior of **continue**, this statement is equivalent to

```
exp-1 ;
while ( exp-2 )
{
        statement
        exp-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second speci-
fies a test, made before each iteration, such that the loop is exited when the
expression becomes 0. The third expression often specifies an incrementing
that is performed after each iteration.

Any or all of the expressions may be dropped.

switch **Statement**

The **switch** statement causes control to be transferred to one of several
statements depending on the value of an expression. It has the form

switch (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result
must be **int**. The statement is typically compound. Any statement within the
statement may be labeled with one or more case prefixes as in the following:

case *constant-expression* :

where the constant expression must be **int**. No two of the case constants in
the same switch may have the same value. Constant expressions are precisely
defined in "Constant Expressions."

There may also be at most one statement prefix of the form

default :

which properly goes at the end of the case constants.

When the **switch** statement is executed, its expression is evaluated and
compared in turn with each case constant. If one of the case constants is
equal to the value of the expression, control is passed to the statement follow-
ing the matched case prefix. If no case constant matches the expression and if
there is a **default** prefix, control passes to the statement prefixed by **default**.
If no case matches and if there is no **default**, then none of the statements in
the switch is executed.

The prefixes **case** and **default** do not alter the flow of control, which continues unimpeded across such prefixes. That is, once a case constant is matched, all **case** statements (and the **default**) from there to the end of the **switch** are executed. To exit from a switch, see "**break** Statement."

Usually, the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective. A simple example of a complete **switch** statement is the following:

```
switch (c) {
        case 'o':
                oflag = TRUE;
                break;
        case 'p':
                pflag = TRUE;
                break;
        case 'r':
                rflag = TRUE;
                break;
        default :
                (void) fprintf(stderr, "Unknown option\n");
                exit(2);
        }
```

break **Statement**

The statement **break ;** causes termination of the smallest enclosing **while**, **do, for**, or **switch** statement; control passes to the statement following the terminated statement.

continue **Statement**

The statement **continue** ; causes control to pass to the loop-continuation portion of the smallest enclosing **while**, **do**, or **for** statement; that is to the end of the loop. More precisely, in each of the statements

```
while (...)       do              for (...)
{                 {               {
    ...               ...             ...
contin: ;         contin: ;       contin: ;
}                 } while (...);   }
```

a **continue** is equivalent to **goto contin**. (Following the **contin:** is a null statement; see "Null Statement.")

return **Statement**

A function returns to its caller by means of the **return** statement, which has one of the forms

> **return ;**
> **return** *expression* ;

In the first case, the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

goto **Statement**

Control may be transferred unconditionally by means of the statement

goto *identifier* ;

The identifier must be a label (see "Labeled Statement") located in the current function.

Labeled Statement

Any statement may be preceded by label prefixes of the form

identifier :

which serve to declare the identifier as a label. The only use of a label is as a target of a **goto**. The scope of a label is the current function, excluding any subblocks in which the same identifier has been redeclared (see "Scope Rules").

Null Statement

The null statement has the form

;

A null statement is useful to carry a label just before the } of a compound statement or to supply a null body to a looping statement such as **while**.

External Definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class **extern** (by default) or perhaps **static**, and a specified type. The type-specifier (see "Type Specifiers" in "Declarations") may also be empty, in which case the type is taken to be **int**. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations except that only at this level may the code for functions be given.

External Function Definitions

Function definitions have the form

> *function-definition:*
> > *decl-specifiers*$_{opt}$ *function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are **extern** or **static**; see "Scope of Externals" in "Scope Rules" for the distinction between them. A function declarator is similar to a declarator for a "function returning ..." except that it lists the formal parameters of the function being defined.

> *function-declarator:*
> > *declarator* (*parameter-list*$_{opt}$)

> *parameter-list:*
> > *identifier*
> > *identifier , parameter-list*

The function-body has the form

> *function-body:*
> > *declaration-list*$_{opt}$ *compound-statement*

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be **int**. The only storage class that may be specified is **register**; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
      int a, b, c;
{
      int m;

      m = (a > b) ? a : b;
      return((m > c) ? m : c);
}
```

Here **int** is the type-specifier; **max(a, b, c)** is the function-declarator;
int a, b, c; is the declaration-list for the formal parameters; { ... } is the block
giving the code for the statement.

The C program converts all **float** actual parameters to **double**, so formal
parameters declared **float** have their declaration adjusted to read **double**. All
char and **short** formal parameter declarations are similarly adjusted to read
int. Also, since a reference to an array in any context (in particular as an
actual parameter) is taken to mean a pointer to the first element of the array,
declarations of formal parameters declared "array of ... " are adjusted to read
"pointer to ... ".

External Data Definitions

An external data definition has the form

    data-definition:
        declaration

The storage class of such data may be **extern** (which is the default) or **static**,
but not **auto** or **register**.

Scope Rules

A C program need not all be compiled at the same time. The source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scopes to consider: first, what may be called the lexical scope of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

Lexical Scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers that are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of a block persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (see "Structure and Union Declarations" and "Enumeration Declarations" in "Declarations") that tag identifiers associated with ordinary variables, and identities associated with structure and union members, form three disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. The **enum** constants are in the same class as ordinary variables and follow the same scope rules. The **typedef** names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
    int distance;
    ...
```

The **int** must be present in the second declaration, or it would be taken to be a declaration with no declarators and type **distance**.

Scope of Externals

If a function refers to an identifier declared to be **extern**, then somewhere among the files or libraries constituting the complete program there must be at least one external definition for the identifier. All functions in a given program that refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function that references the data.

It is illegal to explicitly initialize any external identifier more than once in the set of files and libraries comprising a multi-file program. It is legal to have more than one data definition for any external non-function identifier; explicit use of **extern** does not change the meaning of an external declaration.

In restricted environments, the use of the **extern** storage class takes on an additional meaning. In these environments, the explicit appearance of the **extern** keyword in external data declarations of identities without initialization indicates that the storage for the identifiers is allocated elsewhere, either in this file or another file. It is required that there be exactly one definition of each external identifier (without **extern**) in the set of files and libraries comprising a mult-file program.

Identifiers declared **static** at the top level in external definitions are not visible in other files. Functions may be declared **static**.

Compiler Control Lines

The C compilation system contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with **#** communicate with this preprocessor. There may be any number of blanks and horizontal tabs between the **#** and the directive, but no additional material (such as comments) is permitted. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect that lasts (independent of scope) until the end of the source program file.

Token Replacement

A control line of the form

> **#define** *identifier token-string*$_{opt}$

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in or at the end of the token-string are part of that string. A line of the form

> **#define** *identifier1(identifier, ...) token-string*$_{opt}$

where there is no space between the first identifier and the **(**, is a macro definition with arguments. There may be zero or more formal parameters. Subsequent instances of the first identifier followed by a **(**, a sequence of tokens delimited by commas, and a **)** are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however, commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Strings and character constants in the token-string are scanned for formal parameters, but strings and character constants in the rest of the program are not scanned for defined identifiers to replace.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued. This facility is most valuable for definition of "manifest constants," as in the following example.

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form

> **#undef** *identifier*

causes the identifier's preprocessor definition (if any) to be forgotten.

If a **#define**d identifier is the subject of a subsequent **#define** with no intervening **#undef**, then the two token-strings are compared textually. If the two token-strings are not identical (all white space is considered as equivalent), then the identifier is considered to be redefined.

File Inclusion

A control line of the form

> **#include** *"filename"*

causes the replacement of that line by the entire contents of the file **filename**. The named file is searched for first in the directory of the file containing the **#include**, and then in a sequence of specified or standard places. Alternatively, a control line of the form

> **#include** <*filename*>

searches only the specified or standard places and not the directory of the **#include**. [How the places are specified is not part of the language. See **cpp**(1) in the *Programmer's Reference Manual* for a description of how to specify additional libraries.]

#includes may be nested.

Conditional Compilation

A compiler control line of the form

#if *restricted-constant-expression*

checks whether the restricted-constant expression evaluates to nonzero. (Constant expressions are discussed in "Constant Expressions"; the following additional restrictions apply here: the constant expression may not contain **sizeof**, casts, or an enumeration constant.)

A restricted-constant expression may also contain the additional unary expression

defined *identifier*

or

defined (*identifier*)

which evaluates to one if the identifier is currently defined in the preprocessor and zero if it is not.

All currently defined identifiers in restricted-constant expressions are replaced by their token-strings (except those identifiers modified by **defined**) just as in normal text. The restricted-constant expression will be evaluated only after all expressions have finished. During this evaluation, all undefined (to the procedure) identifiers evaluate to zero.

A control line of the form

#ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; i.e., whether it has been the subject of a **#define** control line. It is equivalent to **#if defined** (*identifier*).

A control line of the form

#ifndef *identifier*

checks whether the identifier is currently undefined in the preprocessor. It is equivalent to **#if !defined** (*identifier*).

All three forms are followed by an arbitrary number of lines, possibly containing a control line

> **#else**

and then by a control line

> **#endif**

If the checked condition is true, then any lines between **#else** and **#endif** are ignored. If the checked condition is false, then any lines between the test and a **#else** or, lacking a **#else**, the **#endif** are ignored.

Another control directive is

> **#elif** *restricted-constant-expression*

An arbitrary number of **#elif** directives can be included between **#if, #ifdef**, or **#ifndef** and **#else**, or **#endif** directives. These constructions may be nested.

Packing Structure Members

When storage is allocated for structures, structure members are ordinarily stored as follows:

■ Items of type **char** or **unsigned char**, or arrays containing items of these types, are byte aligned.

■ Structures are word aligned; structures of odd size are padded to an even number of bytes.

■ All other types of structure members are word aligned.

To conserve space, or to conform to existing data structures, you may want to store structures more or less compactly. The **pack** pragma controls how structure data are "packed" into memory.

Use the **pack** pragma when you want to specify packing other than the packing specified on the command line for particular structures. Use a control line of the form

> **#pragma pack (n)**

where *n* is 1, 2, or 4, before structures that you want to pack differently.

To reinstate the packing given on the command line, give the **pack()** pragma with no arguments.

Line Control

For the benefit of other preprocessors that generate C programs, a line of the form

> **#line** *constant* *"filename"*

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant, and the current input file is named by *"filename"*. If *"filename"* is absent, the remembered file name does not change.

Version Control

This capability, known as *S-lists*, helps administer version control information. A line of the form

> **#ident** *"version"*

puts any arbitrary string in the **.comment** section of the **a.out** file. It is usually used for version control. It is worth remembering that **.comment** sections are not loaded into memory when the **a.out** file is executed.

Types Revisited

This part summarizes the operations that can be performed on objects of certain types.

Structures and Unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right of the -> or the . must specify a member of the aggregate named or pointed to by the expression on the left. In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee is made by the language in order to simplify the use of unions: if a union contains several structures that share a common initial sequence and if the union currently contains one of these structures, it is permitted to inspect the common initial part of any of the contained structures. For example, the following is a legal fragment:

```
union
{
        struct
        {
                int        type;
        } n;
        struct
        {
                int        type;
                int        intnode;
        } ni;
        struct
        {
                int        type;
                float      floatnode;
        } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
        ... sin(u.nf.floatnode) ...
```

Functions

There are only two things that can be done with a function: call it or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

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

Then the definition of **g** might read

```
g(funcp)
        int (*funcp)();
{
        ...
        (*funcp)();
        ...
}
```

Notice that **f** must be declared explicitly in the calling routine, since its appearance in **g(f)** was not followed by **(**.

Arrays, Pointers, and Subscripting

Every time an identifier of array type appears in an expression (except as an operand of "sizeof"), it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not **lvalues**. By definition, the subscript operator **[]** is interpreted in such a way that **E1[E2]** is identical to ***((E1)+(E2))**. Because of the conversion rules that apply to +, if **E1** is an array and **E2** an integer, then **E1[E2]** refers to the **E2**[th] member of **E1**. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multidimensional arrays. If **E** is an n-dimensional array of rank i×j×...×k, then **E** appearing in an expression is converted to a pointer to an $(n-1)$ dimensional array with rank j×...×k. If the * operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$ dimensional array, which itself is immediately converted into a pointer.

For example, consider **int x[3][5];** Here **x** is a 3×5 array of integers. When **x** appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression **x[i]**, which is equivalent to ***(x+i)**, **x** is first converted to a pointer as described; then **i** is converted to the type of **x**, which involves multiplying **i** by the length of the object to which the pointer points, namely 5-integer objects. The results are added and indirection applied to yield an array (of five integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript, the same argument applies again; this time the result is an integer.

Arrays in C are stored row-wise (last subscript varies fastest) and the first subscript in the declaration helps determine the amount of storage consumed by an array. Arrays play no other part in subscript calculations.

Explicit Pointer Conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, see "Unary Operators" under "Expressions" and "Type Names" under "Declarations."

A pointer may be converted to any of the integral types large enough to hold it. Whether an **int** or **long** is required is machine-dependent. The mapping function is also machine-dependent but is intended to be unsurprising to those who know the addressing structure of the machine.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer, but is otherwise machine-dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a **char** pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

The **alloc** must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to **double**; then the use of the function is portable.

Constant Expressions

In several places C requires expressions that evaluate to a constant: after **case**, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, casts to integral types, enumeration constants, and **sizeof** expressions, possibly connected by the binary operators

$$+ - * / \% \& |\ \char`^\ << >> == != < > <= >= \&\& ||$$

or by the unary operators

$$- \char`~$$

or by the ternary operator

$$?:$$

Parentheses can be used for grouping but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also use floating constants and arbitrary casts and can also apply the unary **&** operator to external or static objects and to external or static arrays subscripted with a constant expression. The unary **&** can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

Portability Considerations

Certain parts of C are inherently machine-dependent. The following list of potential trouble spots is not meant to be all-inclusive but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are nuisances that must be carefully watched. Most of the others are only minor problems.

The number of **register** variables that can actually be placed in registers varies from machine to machine as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid **register** declarations are ignored.

The order of evaluation of function arguments is not specified by the language. The order in which side effects take place is also unspecified.

Since character constants are really objects of type **int**, multicharacter character constants may be permitted. The specific implementation is very machine-dependent because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right to left on some machines and left to right on other machines. These differences are invisible to isolated programs that do not indulge in type punning (e.g., by converting an **int** pointer to a **char** pointer and inspecting the pointed-to storage) but must be accounted for when conforming to externally-imposed storage layouts.

Syntax Summary

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

Expressions

The basic expressions are:

expression:
> *primary*
> * *expression*
> & *lvalue*
> − *expression*
> ! *expression*
> ~ *expression*
> ++ *lvalue*
> −− *lvalue*
> *lvalue* ++
> *lvalue* −−
> **sizeof** *expression*
> **sizeof** (*type-name*)
> (*type-name*) *expression*
> *expression binop expression*
> *expression ? expression : expression*
> *lvalue asgnop expression*
> *expression , expression*

primary:
> *identifier*
> *constant*
> *string literal*
> (*expression*)
> *primary (expression-list$_{opt}$)*
> *primary [expression]*
> *primary . identifier*
> *primary −> identifier*

> *lvalue:*
> > *identifier*
> > *primary [expression]*
> > *lvalue . identifier*
> > *primary -> identifier*
> > * *expression*
> > *(lvalue)*

The primary-expression operators

> () [] . ->

have highest priority and group left to right. The unary operators

> * & – ! ~ ++ –– **sizeof**   *(type-name)*

have priority below the primary operators but higher than any binary operator and group right to left. Binary operators group left to right; they have priority decreasing as indicated below.

> *binop:*
> > *   /   %
> > +   –
> > >>   <<
> > <   >   <=   >=
> > ==   !=
> > &
> > ^
> > |
> > &&
> > ||

The conditional operator groups right to left.

   Assignment operators all have the same priority and all group right to left.

> *asgnop:*
> > = += –= *= /= %= >>= <<= &= ^= |=

The comma operator has the lowest priority and groups left to right.

Declarations

declaration:
  decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:
  type-specifier decl-specifiers$_{opt}$
  sc-specifier decl-specifiers$_{opt}$

sc-specifier:
  auto
  static
  extern
  register
  typedef

type-specifier:
  struct-or-union-specifier
  typedef-name
  enum-specifier

basic-type-specifier:
  basic-type
  basic-type basic-type-specifiers

basic-type:
  char
  short
  int
  long
  unsigned
  float
  double
  void

enum-specifier:
>   **enum** { *enum-list* }
>   **enum** *identifier* { *enum-list* }
>   **enum** *identifier*

enum-list:
>   *enumerator*
>   *enum-list , enumerator*

enumerator:
>   *identifier*
>   *identifier = constant-expression*

init-declarator-list:
>   *init-declarator*
>   *init-declarator , init-declarator-list*

init-declarator:
>   *declarator initializer$_{opt}$*

declarator:
>   *identifier*
>   *(declarator)*
>   ** declarator*
>   *declarator ()*
>   *declarator [constant-expression$_{opt}$]*

struct-or-union-specifier:
>   **struct** { *struct-decl-list* }
>   **struct** *identifier* { *struct-decl-list* }
>   **struct** *identifier*
>   **union** { *struct-decl-list* }
>   **union** *identifier* { *struct-decl-list* }
>   **union** *identifier*

struct-decl-list:
>   *struct-declaration*
>   *struct-declaration struct-decl-list*

struct-declaration:
    type-specifier struct-declarator-list ;

struct-declarator-list:
    struct-declarator
    struct-declarator , struct-declarator-list

struct-declarator:
    declarator
    declarator : constant-expression
    : constant-expression

initializer:
    = expression
    = { initializer-list }
    = { initializer-list , }

initializer-list:
    expression
    initializer-list , initializer-list
    { initializer-list }
    { initializer-list , }

type-name:
    type-specifier abstract-declarator

abstract-declarator:
    empty
    (abstract-declarator)
    * abstract-declarator
    abstract-declarator ()
    abstract-declarator [constant-expression$_{opt}$]

typedef-name:
    identifier

Statements

compound-statement:
    { declaration-list$_{opt}$ statement-list$_{opt}$ }

declaration-list:
    declaration
    declaration declaration-list

statement-list:
    statement
    statement statement-list

statement:
    compound-statement
    expression ;
    if (expression) statement
    if (expression) statement **else** statement
    while (expression) statement
    do statement **while** (expression) ;
    for (exp$_{opt}$; exp$_{opt}$; exp$_{opt}$) statement
    switch (expression) statement
    case constant-expression : statement
    default : statement
    break ;
    continue ;
    return ;
    return expression ;
    goto identifier ;
    identifier : statement
    ;

External Definitions

program:
  external-definition
  external-definition program

external-definition:
  function-definition
  data-definition

function-definition:
  decl-specifier$_{opt}$ function-declarator function-body

function-declarator:
  declarator (parameter-list$_{opt}$)

parameter-list:
  identifier
  identifier , parameter-list

function-body:
  declaration-list$_{opt}$ compound-statement

data-definition:
  extern *declaration ;*
  static *declaration ;*

Preprocessor

#define *identifier token-string$_{opt}$*
#define *identifier(identifier,...)token-string$_{opt}$*
#undef *identifier*
#include *"filename"*
#include *<filename>*
#if *restricted-constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#elif *restricted-constant-expression*
#else
#endif
#line *constant "filename"*
#ident *"version"*
#pragma pack *([1/2/4])*

18 C Programmer's Productivity Tools

Introducing the C Programmer's Productivity Tools

This chapter will teach you how to use the C Programmer's Productivity Tools (CPPT). First, step by step instructions are provided in the context of basic examples so you can start using CPPT right away. Additional examples demonstrate various options that allow you to make the best use of the tools. To use CPPT you must know how to use the other tools in the C Software Development Set.

The CPPT package consists of two tools: **cscope** and **lprof**. **cscope** is a browser; **lprof**, a profiler.

The **cscope** browser is an interactive program that locates specified parts of code in a set of C source files and gives you the option of editing those files. It can significantly reduce the amount of time you must spend searching for functions, function calls, macros, and variables in the code. Programmers responsible for writing programs (especially large ones) or maintaining existing programs will be able to edit their source code more efficiently with **cscope**. It is especially helpful for a programmer working on someone else's code. The section **"cscope"** is a tutorial on using this browser.

A profiler is a tool for analyzing a program's run-time behavior, a procedure known as dynamic analysis. **lprof** allows a programmer or tester to identify those parts of the source code that are most often executed and those that are never executed when a program is run. **lprof** provides line by line frequency profiling, reporting how many times each line of source code is executed. The **–x** option allows you to request test coverage analysis so that **lprof** reports only which lines of code are not actually executed at run time. It can be used over a set of tests such as are included in a test suite. The section **"lprof"** teaches you how to use this profiler to perform these types of dynamic analysis.

The section **"Profiling Examples"** presents examples of how profiling can be used to improve program performance. To enhance the effectiveness of **lprof**, use of another profiler, **prof**, is recommended. **prof** reports the amount of time spent in various portions of a program. Once this has been determined, **lprof** can be used to obtain line specific information about the heavily executed portions of code identified by **prof**. These lines can then be rewritten to execute more efficiently.

> [NOTE] **prof** is available in CPLU Issues 3 and 3.1, and C-FP+. For CPLU Issue 4 and subsequent issues, **prof** is included in the Advanced Programming Utilities (APU) package.

lprof performs the following functions:

■ produces source listings

■ produces summary reports of profile data

■ merges profile data files

> [NOTE] The text in this chapter was prepared with UNIX System text editors and formatted with the DOCUMENTER'S WORKBENCH Software: the **troff**, **tbl**, and **mm** macros.

cscope

How cscope Works

Imagine you arrive at work one day and are asked to learn how a particular program works. You are given a large stack of source code printouts and a cross-reference table for them. How do you go about studying the code? Until now, programmers have had to flip back and forth through pages of printouts to find the functions, function calls, macros, and variables listed in the cross-reference.

Now, however, you can use an interactive electronic tool to search through the code for you. This tool is the **cscope** browser. **cscope** builds a cross-reference symbol table for the functions, function calls, macros, and variables in the source files you specify. It then allows you to query that table about the locations of symbols you specify. Specifically, **cscope** presents a menu and asks you to choose the type of search you would like to have performed. For example, you may want **cscope** to find all functions that call a particular function.

When **cscope** has completed this search, it prints a list of the lines on which it has found the item (such as the functions calling a function) that you specified. It then waits for you to specify which of these lines you want to examine. After you have requested a subset of the lines **cscope** waits for you to edit a line (by using the default editor, **vi**, or an editor of your choice) or to begin another search.

Throughout a **cscope** session, you have the option of returning to the menu from the editor to request a new search. There are a variety of single-character commands available for manipulating the menu.

Because the procedure you follow will depend on the task you select, there is no single set of instructions for using **cscope**. To learn how this browser works, study the following example. It shows how you can locate a bug in a program without learning all the code.

Step 1: Identify the Problem

Suppose you are responsible for maintaining **cscope** itself. You notice that an error message, out of storage, sometimes appears when you run the program. How can you fix this? First, locate the parts of the code that are

generating the message. Use **cscope** to find these parts quickly.

Step 2: Set Up the Environment

cscope uses an editor as the medium through which you browse through your files. Therefore, before installing CPPT, you must check your environment to be sure an editor that can be used on your terminal is accessible.

Check the value of the TERM environment variable to make sure you have set it for your terminal. The first time you logged in you should have done this by assigning a value to TERM and exporting TERM to the shell, as follows:

```
$ TERM=term_name
$ export TERM
```

You may now want to assign a value to the EDITOR environment variable. By default, **cscope** invokes the **vi** editor. If you prefer not to use **vi**, set the EDITOR environment variable to the editor of your choice and export EDITOR. (See the "Command Line Syntax for Editors" section under "Cautionary Notes on using **cscope**" for details and examples.)

If you want to use **cscope** only for browsing (without editing) you can set the VIEWER environment variable to **pg** and export VIEWER. **cscope** will then invoke **pg** instead of **vi**.

NOTE	Using the **ed** or **jim** editor is possible but not recommended. (**jim** is an editor that takes advantage of the multi-screen capability of the AT&T 5620 terminal. It cannot be used with any other type of terminal.) See "Cautionary Notes on Using **cscope**" for suggested workarounds for these editors.

Once you have set up your environment so that **cscope** will call the editor of your choice, you are ready to use the browser.

Step 3: Invoke cscope

If all the source files for the program to be browsed (with the possible exception of standard system header files) are in the current directory, invoke **cscope** without any arguments:

```
$ cscope
```

By default, **cscope** builds its cross-reference table for all the C, **lex**, and **yacc**

source files in the current directory. Therefore, typing **cscope** without any arguments is equivalent to the following command line:

$ **cscope *.[chly]**

cscope will also search the standard directories for any header files that you include with **#include**.

NOTE | For other ways to invoke **cscope** (including a way to invoke it if the source files are in multiple directories) see "Other Command Line Options" later in this section.

The Cross-Reference File

When **cscope** is first invoked, it builds a cross-reference symbol table to which it refers during subsequent sessions. This table is created in the current directory and is called **cscope.out**. The next time **cscope** is invoked, it checks **cscope.out** for changes. **cscope** modifies the table if the list of source files has been changed. Also, if the table has been modified, **cscope** rebuilds only those portions of the table that have been modified. Because copying information is much faster than building it, subsequent calls to **cscope** should require much less start-up time than the initial call.

Running cscope

After **cscope** has been invoked and the cross-reference information processed, the **cscope** menu of tasks appears on the screen.

```
  cscope              Press the ? key for help




















  List references to this C symbol:
  Edit this function or #define:
  List functions called by this function:
  List functions calling this function:
  List lines containing this text string:
  Change this text string:
  List file names containing this text string:
```
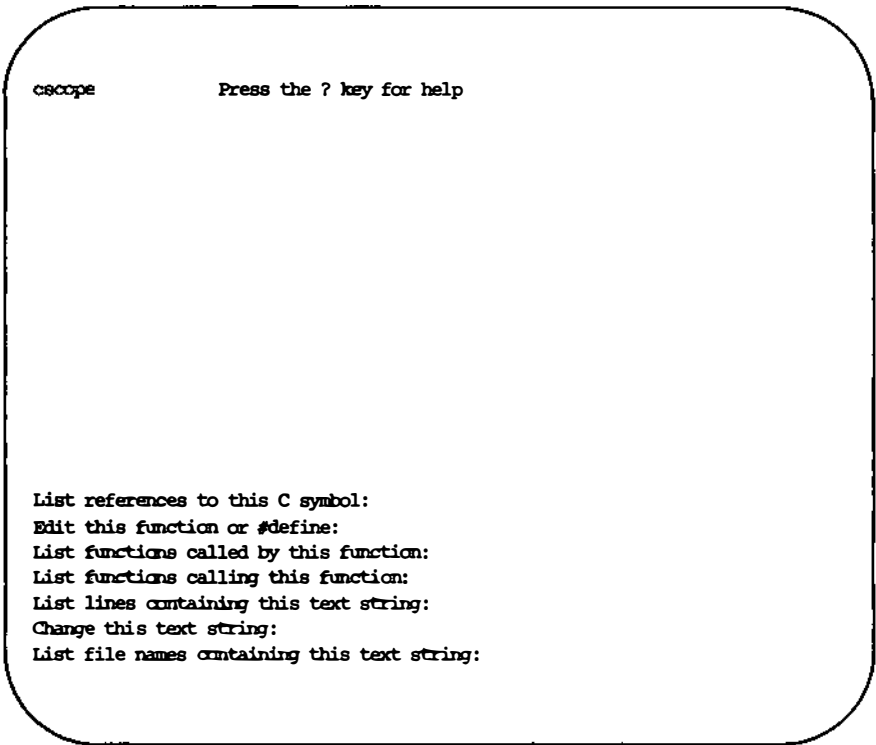
Figure 18-1: The **cscope** Menu of Tasks

Press the TAB or RETURN key to move the cursor down the screen (with
wraparound at the bottom of the display), and ^p (control-p) to move the cur-
sor up. Once the cursor is at the desired input field, enter the text to be
searched, and press the RETURN key.

The following single-key commands are available at any time during a
cscope session.

> **NOTE** The ^ (circumflex) represents the CONTROL key. Instructions to type control characters (such as ^p in the previous paragraph) should be followed by holding down the CONTROL key and pressing the letter shown after the circumflex.

TAB	move to next input field
RETURN	move to next input field
^m	move to next input field
^p	move to previous input field
.	search with the last text typed
^r	rebuild the cross-reference
!	start an interactive shell (type ^d to return to **cscope**)
^l	redraw the screen
?	display list of commands
^d	exit **cscope**

Figure 18-2: Menu Manipulation Commands

Step 4: Locate the Source of the Error Message

Now let's return to the task we undertook at the beginning of the section **cscope**: to fix the problem that is causing the error message out of storage to be printed. You have invoked **cscope**; the menu is on the screen. Start your search for the problem by locating the section of code where the error message is generated. Move the cursor to the fifth menu item (List lines containing this text string) and enter the text **out of storage**.

```
┌─────────────────────────────────────────────────────────────┐
│                                                             │
│  cscope          Press the ? key for help                   │
│                                                             │
│                                                             │
│                                                             │
│                                                             │
│                                                             │
│                                                             │
│                                                             │
│                                                             │
│  List references to this C symbol:                          │
│  Edit this function or #define:                             │
│  List functions called by this function:                    │
│  List functions calling this function:                      │
│  List lines containing this text string: out of storage     │
│  Change this text string:                                   │
│  List file names containing this text string:               │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```
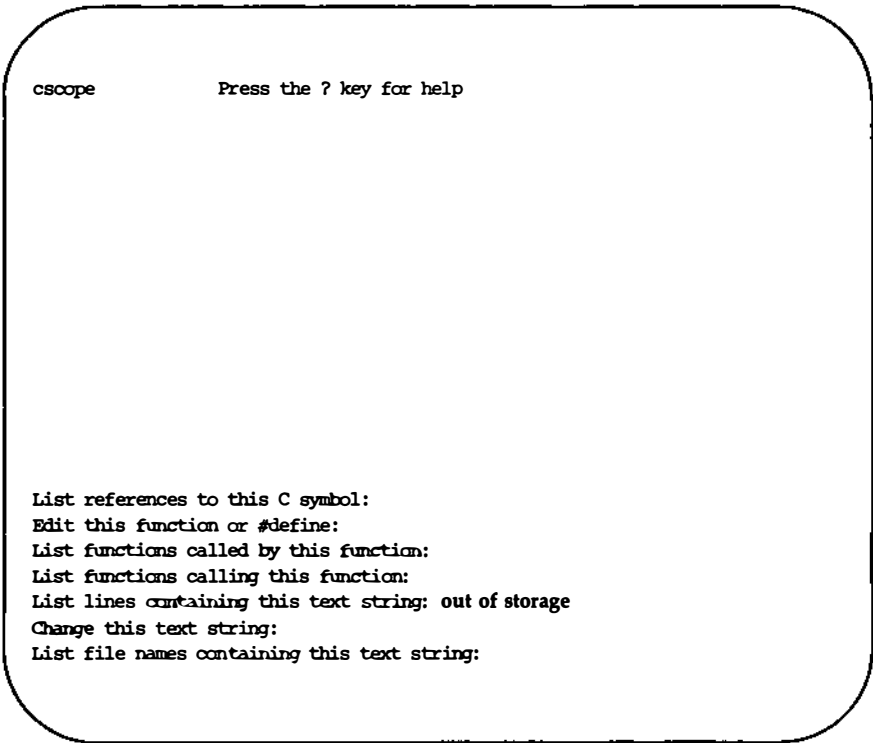
Figure 18-3: Requesting a Search for a Text String

Press the RETURN key. **cscope** searches for the specified text and finds one line that contains it.

┌──────┐ Follow the same procedure to perform any other task listed in the menu
│ NOTE │ except the sixth, Change this text string. Because this task is slightly
└──────┘ more complex than the others, there is a different procedure for performing
          it. For a description and examples of changing a text string, see "Examples
          of Using **cscope**" later in this section.

cscope reports its finding as follows:

```
Text string: out of storage

 File    Line
 1 alloc.c 56 (void) fprintf(stderr, "\n%s: out of storage\n",
 argv0);

















 List references to this C symbol:
 Edit this function or #define:
 List functions called by this function:
 List functions calling this function:
 List lines containing this text string:
 Change this text string:
 List file names containing this text string:
```

Figure 18-4: **cscope** Lists Lines Containing the Text String

After **cscope** shows you the results of a successful search in this way, you have several options. For example, you may want to edit one of the lines found. Or, if **cscope** has found so many lines that a list of them will not fit on the screen at once, you may want to look at the next part of the list. The following table shows the commands available after **cscope** has found the specified text.

1-9	edit this line (the number you type corresponds to an item in the list of lines printed by **cscope**)
space	display the lines after the current line
+	display the lines after the current line
-	display the lines before the current line
^e	edit all lines
>	append the list of lines being displayed to a file

Figure 18-5: Commands for Use After Initial Search

If the first character of the text for which you are searching matches one of these commands, be sure to precede it with a \ (backslash).

Now examine the code around the newly found line. Enter **1** (the number of the line in the list). The editor will be invoked with the file **alloc.c**; the cursor will be at the beginning of line 56 of the text file.

```
{
        return(alloctest(realloc(p, (unsigned) size)));
}

/* check for memory allocation failure */

static  char *
alloctest(p)
char    *p;
{
        if (p == NULL) {
                (void) fprintf(stderr, "\n%s: out of storage\n",
                argv0);
                exit(1);
        }
        return(p);
}
~
~
~
~
~
~
~
"alloc.c" 60 lines, 1022 characters
```

Figure 18-6: Examining a Line of Code Found by **cscope**

By examining the code, you notice that the error message is generated when the variable *p* is NULL. To determine how an argument passed to **alloctest** could have been NULL, you must first identify the functions that call **alloctest**.

Exit the editor by using normal write and quit conventions, and return to the menu of tasks. Now type **alloctest** after the fourth item, List functions calling this function.

```
Text string: out of storage

 File    Line
1 alloc.c 56 (void) fprintf(stderr, "\n%s: out of storage\n",
argv0);











List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function: alloctest
List lines containing this text string:
Change this text string:
List file names containing this text string:
```

Figure 18-7: Requesting a List of Functions that Call **alloctest**

cscope finds and lists three such functions.

```
Functions calling this function: alloctest

  File     Function  Line
1 alloc.c mymalloc  26 return(alloctest(malloc((unsigned) size)));
2 alloc.c mycalloc  36 return(alloctest(calloc((unsigned) nelem,
  (unsigned)
                        size)));
3 alloc.c myrealloc 46 return(alloctest(realloc(p, (unsigned)
  size)));




















List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string:
List file names containing this text string:
```

Figure 18-8: **cscope** Lists Functions that Call **alloctest**

Now you want to know which functions call **mymalloc**. **cscope** finds ten such functions. It lists seven of them on the screen and instructs you to press the space bar to see the rest of the list.

```
Functions calling this function: mymalloc

  File     Function       Line
1 alloc.c  stralloc       17 return(strcpy(mymalloc(strlen(s)
                          + 1), s));
2 dir.c    makesrcdirlist 70 srcdirs = (char **)
                          mymalloc(nsrcdirs * sizeof(char *));
3 dir.c    makesrcdirlist 89 s = mymalloc(strlen(srcdirs[i])
                          + n);
4 dir.c    makefilelist   115 srcfiles = (char **)
                          mymalloc(msrcfiles * sizeof(char *));
5 dir.c    makefilelist   116 srcnames = (char **)
                          mymalloc(msrcfiles * sizeof(char *));
6 dir.c    addincdir      212 incdirs = (char **)
                          mymalloc(sizeof(char *));
7 display.c dispinit      76 displine = (int *)
                          mymalloc(mdisprefs * sizeof(int));

* 3 more lines - press the space bar to display more *
List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string:
List file names containing this text string:
```

Figure 18-9: **cscope** Lists Functions that Call **mymalloc**

Because you know that the error message (out of storage) is generated at the beginning of the program, you can guess that the problem may have occurred in the function **dispinit** (display initialization). To view **dispinit**, the seventh function on the list, type **7**.

```
void
dispinit()
{
        /* calculate the maximum displayed reference lines */
        lastdispline = FLDLINE - 4;
        mdisprefs = lastdispline - REFLINE + 1;
        if (mdisprefs > 9) {
                mdisprefs = 9;
        }
        /* allocate the displayed line array */
        displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
{
        char    file[PATHLEN + 1];      /* file name */
        char    function[PATLEN + 1];   /* function name */
        char    linenum[NUMLEN + 1];    /* line number */
        int     screenline;             /* screen line number */
        int     width;                  /* source line display
                                           width */
        register int    i, j;
"display.c" 440 lines, 10198 characters
```

Figure 18-10: Viewing **dispinit** in the Editor

mymalloc failed because it was called with either a very large number or with a negative number. By examining the possible values of FLDLINE and REFLINE, you can see that there are situations in which the value of the variable is negative, that is, in which you are trying to call **mymalloc** with a negative number. The program needs a mechanism so that if the value of the variable is negative, it will abort after printing a meaningful error message.

For example, on an AT&T 5620 terminal you may have multiple windows of arbitrary size. The error message might appear as a result of running **cscope** in a layer that has too few lines. One solution to this problem is to have the program print an error message stating that the screen is too small. Edit the function **dispinit** as follows:

```
/* initialize display parameters */

void
dispinit()
{
        /* calculate the maximum displayed reference lines */
        lastdispline = FLDLINE - 4;
        mdisprefs = lastdispline - REFLINE + 1;
        if (mdisprefs <= 0) {
                (void) fprintf(stderr,"\n%s: screen too small\n",
                argv0);
                exit(1);
        }
        if (mdisprefs > 9) {
                mdisprefs = 9;
        }
        /* allocate the displayed line array */
        displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
```

Figure 18-11: Using **cscope** to Fix the Problem

You have now fixed the problem we began investigating at the beginning of this section. If the screen is not large enough when you run your program in the future, the program will not simply fail with the cryptic error message out of storage. Instead, it will check the screen size and generate a more meaningful error message before exiting.

Other Command Line Options

cscope examines all the C, **lex**, and **yacc** source files in the current direc-
tory by default. Thus

    $ cscope

is equivalent to

    $ cscope *.[chly]

The **cscope** command line provides several options that allow the pro-
grammer greater flexibility in selecting source files to be included in the
cross-reference. To browse through specific files, invoke **cscope** with file
names as arguments on the command line:

    $ cscope file1.c file2.c file3.h

To specify a file containing a list of all the files to be browsed, use the **–i**
option. If the source is in a directory tree, the following commands will allow
you to examine all the source files easily:

    $ find . –name '*.[chly]' –print ¦ sort > *filelist*
    $ cscope –i *filelist*

The **–I** option for **cscope** is similar to the **–I** option for **cc**. It directs
cscope to search specified directories for **#include** files.

    $ cscope –I../hdr

cscope automatically searches for the **#include** files that it encounters in the
files it scans.

The programmer can specify a cross-reference file other than **cscope.out**
by using the **–f** option. This is useful for keeping separate symbol cross-
reference files in the same directory. A programmer may want to do this if
two programs are in the same directory, but do not share all the same files.

    $ cscope –f admin.ref admin.c common.c aux.c libs.c
    $ cscope –f delta.ref delta.c common.c aux.c libs.c

In the preceding example, the source for two programs (**admin** and **delta**) are in the same directory, but the programs comprise different files. Suppose you are running **cscope** on **admin**. You may not want to see references to symbols in **delta.c**. By specifying two reference files, the cross-reference information for the two programs can be kept separate.

As with **cscope.out**, if the alternate file does not exist, **cscope** will build the cross-reference and leave it in the file specified by the **-f** option.

cscope offers an option, **-d**, that allows you to prevent updating of the cross-reference table and thereby save time. It is permissible to use this option if you are sure that your source files have not been changed. However, because it is usually more important to safeguard against generating erroneous data than to save time, avoid using the **-d** option unless absolutely necessary.
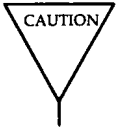
Use the **-d** option with extreme caution. If you specify **-d** with **cscope** under the erroneous impression that your source files have not been changed, **cscope** will give you data for an outdated program.

Optional Features

This section describes some of the more advanced capabilities of **cscope**.

Stacking cscope **and Editor Calls**

cscope and editor calls can be stacked. This means that when **cscope** puts you in the editor to display one symbol reference and there is another symbol of interest, you can call **cscope** again from within the editor without exiting the current invocation of either **cscope** or the editor. You can then back up to a previous invocation by exiting the appropriate **cscope** and editor calls.

Directories Searched by cscope

cscope searches for header files in the following directories in this order:

1. the current directory

2. directories specified by the **-I** option (if they exist)

3. the standard location for header files (usually **usr/include**).

cscope searches for source files only in the current directory.

Using Viewpaths

The environment variable VPATH replaces the current directory in the order of directories searched by **cscope**. This enables you to extend your search for source files from a single directory to a set of directories.

| NOTE | You must specify your current directory in VPATH if you want it to be searched. (The current directory can always be represented by the . symbol.) |

To set VPATH, list the directories you want searched (by their path names) in the order you want them searched. Separate each directory name with colons.

For example, suppose you have a program that consists of three files, **a.c**, **b.c**, and **c.c**. You have assigned the following directories to the VPATH variable:

$ **VPATH=/fs2/mydirectory:/fs1/delivered:/fs1/proj/official**

cscope first searches for the file **a.c** in the directory **/fs1/mydirectory**. If the file is not in that directory **cscope** continues searching for it in the other directories specified in VPATH until it finds the file. Similarly, **cscope** searches the directories in the specified order for **b.c** and **c.c**.

Examples of Using cscope

This section presents examples of how **cscope** can be used to perform three tasks: change a constant to a preprocessor symbol, add an argument to a function, and change the value of a variable.

Changing a Text String

The standard procedure for calling tasks listed on the **cscope** menu of tasks was described in "Running **cscope**" under "Step 3: Invoke **cscope**" early in this section. One task on the menu differs slightly from the others and necessitates following a different procedure.

If you select the sixth menu item, Change this text string, **cscope** will prompt you for new text and then display the lines containing the old text. You can select the lines you want changed with any of the following single-key commands:

1-9	mark or unmark the line to be changed
*	mark or unmark all displayed lines to be changed
space	display next lines
+	display next lines
-	display previous lines
a	mark all lines to be changed
^d	change the marked lines and exit
ESC	exit without changing the marked lines

Figure 18-12: Commands for Selecting Lines to be Changed

The rest of this section consists of more detailed examples.

Changing a Constant to a Preprocessor Symbol

Suppose you want to change a constant, 100, to a preprocessor symbol, MAXSIZE. Select the sixth menu item (Change this text string) and enter \100.

> **NOTE** The 1 must be preceded with a \ (backslash) because it has a special meaning (item 1 on the menu) to cscope.

Now press RETURN; cscope will prompt you for the new text string. Type MAXSIZE.

```
  cscope                                    Press the ? key for help




  List references to this C symbol:
  Edit this function or #define:
  List functions called by this function:
  List functions calling this function:
  List lines containing this text string:
  Change this text string: 100
  List file names containing this text string:
  To: MAXSIZE
```
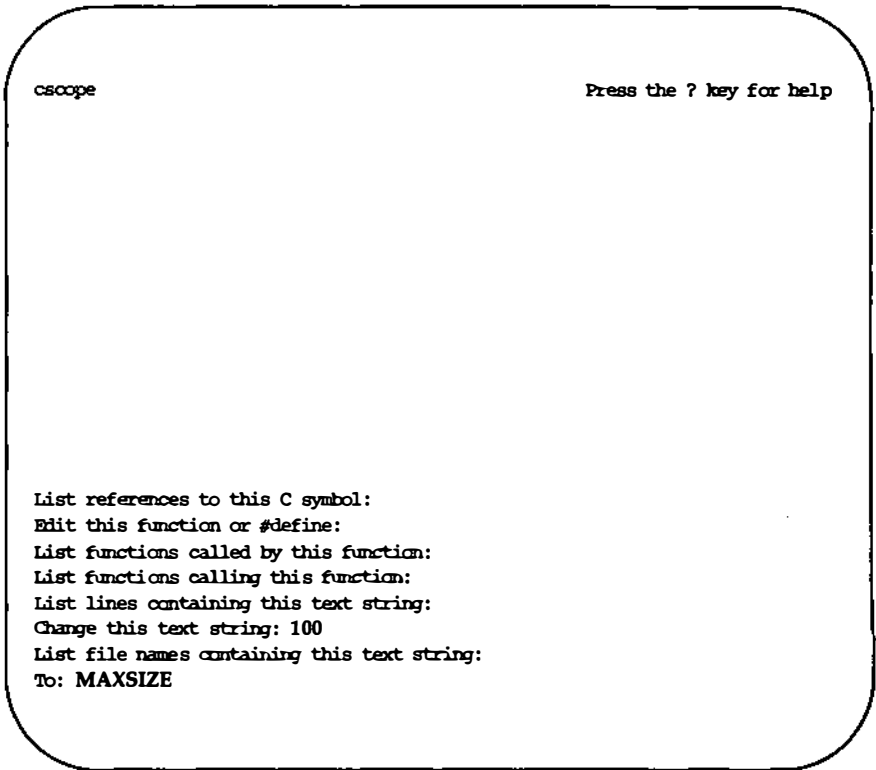
Figure 18-13: Changing a Text String

cscope then displays the lines containing the specified text string, and waits for you to specify the subset of these lines in which you want the text to be changed.

```
Change "100" to "MAXSIZE"

  File  Line
1 init.c   4 char s[100];
2 init.c  26 for (i = 0; i < 100; i++)
3 find.c   8 if (c < 100) {
4 read.c  12 f = (bb & 0100);
5 err.c   19 p = total/100.0; /* get percentage */




List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string:
List file names containing this text string:
Select lines to change (press the ? key for help):
```

Figure 18-14: **cscope** Prompts for Lines to be Changed

You know that occurrences of 100 in lines 1, 2, and 3 of the list (from lines 4, 26, and 8 of the program) are to be changed to MAXSIZE. However, the occurrences of 100 in **read.c** and **err.c** (lines 4 and 5 of the list) are not related; in these lines, 100 should not be changed. Enter **1, 2,** and **3.**

The numbers you type are not printed on the screen. Instead, **cscope** prints a > (greater than) symbol after each number of the list that you type. For example, after you type **1,** a > symbol is printed after the number 1 (and before the line init.c 4 char s[100];) in the list, as shown in Figure 18-15.

```
Change "100" to "MAXSIZE"

  File  Line
1>init.c   4 char s[100];
2>init.c  26 for (i = 0; i < 100; i++)
3>find.c   8 if (c < 100) {
4 read.c  12 f = (bb & 0100);
5 err.c   19 p = total/100.0; /* get percentage */




List references to this C symbol:
Edit this function or #define:
List functions called by this function:
List functions calling this function:
List lines containing this text string:
Change this text string:
List file names containing this text string:
Select lines to change (press the ? key for help):
```

Figure 18-15: Marking Lines to be Changed

After selecting lines, type ˆd to change them. **cscope** then displays the lines that have been changed.

```
Changed lines:

    char s[MAXSIZE];
    for (i = 0; i < MAXSIZE; i++)
        if (c < MAXSIZE) {

Type any character to continue:
```

Figure 18-16: **cscope** Displays Changed Lines of Text

When you type a character in response to this prompt, **cscope** will pause and redraw the screen before allowing you to continue with the session, as shown in Figure 18-17.

The next step is to add the **#define** for the new symbol MAXSIZE. Escape to the shell by typing !. (The shell prompt will appear at the bottom of the screen.) Then enter the editor and add the **#define**.

```
┌─────────────────────────────────────────────────┐
│                                                   │
│  Text string: 100                                 │
│                                                   │
│   File  Line                                      │
│  1 init.c  4 char s[100];                         │
│  2 init.c 26 for (i = 0; i < 100; i++)            │
│  3 find.c  8 if (c < 100) {                       │
│  4 read.c 12 f = (bb & 0100);                     │
│  5 err.c  19 p = total/100.0; /* get percentage */│
│                                                   │
│                                                   │
│                                                   │
│                                                   │
│                                                   │
│                                                   │
│  List references to this C symbol:                │
│  Edit this function or #define:                   │
│  List functions called by this function:          │
│  List functions calling this function:            │
│  List lines containing this text string:          │
│  Change this text string:                         │
│  List file names containing this text string:     │
│  $ vi defs.h                                      │
│                                                   │
└─────────────────────────────────────────────────┘
```

Figure 18-17: Escaping from **cscope** to the Shell

To resume the **cscope** session, quit the editor and type ˆ**d** to exit the shell.

Adding an Argument to a Function

cscope makes it easy to add an argument to a function. Adding an argument involves two steps: editing the function itself and adding the new argument to each place where the function is called.

First, edit the function by using the second menu item, **Edit this function or #define**. Next, find out where the function is called. By invoking the fourth menu item, **List functions calling this function**, you can get a list of all functions that call it. With this list, you can either invoke the editor on each line found by entering the list number for each line individually, or

invoke the editor on all lines automatically by typing ^e. Using **cscope** to make this kind of change is especially useful because it guarantees that none of the functions you need to edit will be overlooked.

Changing the Value of a Variable

The value of **cscope** as a browser becomes apparent when you want to see how a proposed change will affect your code. Suppose you want to change the value of a variable or preprocessor symbol. Before doing so, use the first menu item (List references to this C symbol) to obtain a list of references that will be affected. Then use the editor to examine each one. This will help you predict the overall effects of your proposed change. Later, you can use **cscope** in this manner again to verify that your changes have been made.

Cautionary Notes on Using cscope

This section describes solutions for several problems that may arise while you are using **cscope**.

Unknown Terminal Type

You may see the following error message:

    cscope: "*term*" is not in the terminal database.

If this message appears, your terminal may not be listed in the terminal information (terminfo) database that is currently loaded. Try reloading the database from the Terminal Information Utilities.

You may also see

    cscope: TERM variable is not set or is not exported in your .profile

If this message appears, set and export the TERM variable as described at the beginning of this section (see "Step 2: Set Up the Environment").

Dumping Core

Your system may dump core if the following sequence of events occurs.

1. You make changes to your source code using **cscope**.

2. You rebuild the cross-reference table using the ^R command. (^R rebuilds the table only if you have made changes.)

3. After the table has been rebuilt, the list of references previously displayed becomes obsolete. The screen is cleared so it resembles the initial screen shown by **cscope** (see Figure 18-1).

4. If you try to append the contents of this screen (nothing) to a file by using the > command, **cscope** will dump core and leave your terminal in an unusable state.

To avoid this situation, make sure you see lines of text displayed before trying to append them to a file.

Command Line Syntax for Editors

By default, **cscope** invokes the **vi** editor. **cscope** expects **vi** and any other editor it uses to have a standard command line syntax of the following form:

editor +linenum filename

If you want to use an editor that has this command line syntax, set the EDITOR environment variable to the editor of your choice and export EDITOR. For example, if you want to use the **emacs** editor enter the following commands:

    $ EDITOR=emacs
    $ export EDITOR

However, if the editor you want to use does not conform to this command line syntax, you must write an interface between **cscope** and the editor.

For example, suppose you want to use **ed**. You have already set the EDITOR variable to **ed** and exported it. However, because the **ed** editor does not allow specification of a line number on the command line, you will not be able to edit or view any files while using **cscope**. To solve this problem, write a shell script called **myedit** that contains the following line:

    /bin/ed $2

Then set the value of EDITOR to your shell script.

    $ EDITOR=myedit

Now when **cscope** invokes the editor, it will call this shell script with the following command line:

    myedit +17 main.c

myedit will discard the line number ($1) and call **ed** correctly with the file name ($2).

> **NOTE**
>
> **ed** has one other drawback as a **cscope** editor that you should take into consideration when selecting an editor: it cannot move you to specified lines in the file. If you use the shell script shown in the previous example, you will have to move to specified lines manually.

Using jim

jim is an editor designed to be used exclusively with the AT&T Model 5620 terminal. The 5620 has a large screen (measuring 8-1/2 by 11 inches) and can hold up to six windows simultaneously. The terminal contains its own processor.

jim takes advantage of the 5620's multi-screening capability. It allows a user to move text among screens, as well as among files, and to perform other tasks not available with **vi** or other editors. However, because **jim** is built around the 5620 software, it must be downloaded into the terminal every time you use it. Because downloading is time consuming and **cscope** evokes the editor frequently, using **jim** as a **cscope** editor is not recommended.

If you want to use **jim** with **cscope**, try loading it into one window of your terminal and using **pg** as the **cscope** viewer in another window. This will obviate the need to download **jim** every time you want to look at a symbol reference.

lprof

Introduction

As described in "Introducing the C Programmer's Productivity Tools," there are two profilers available for dynamic analysis of C programs written in a UNIX System environment.

- **prof** performs time profiling; it reports how much time is spent executing various portions of a program.

- **lprof** performs line by line frequency profiling; it reports how many times each line of source code is executed.

| NOTE | The **prof** command is available with Issues 3 and 3.1 of CPLU, but not with Issue 4 or later releases. For those using Issue 4 or a later release, **prof** is available in APU. **prof** is also included with C-FP+. |

To use either of these profilers, you must follow a three-step procedure.

Step 1:   Compile your program with a profiling option.

    for **prof**:   **cc -qp** (or **-p**)
    for **lprof**:   **cc -ql**

Step 2:   Run the profiled program so that run-time data can be collected. At the end of execution the run-time data is written to another file known as a data file. A data file consists of a header section, followed by a section for each function and an end-of-data marker at the end of the file. The coverage data (execution count) for each function is recorded alongside the function's name.

Data files have the following default names

    for **prof**:   **mon.out**
    for **lprof**:   *prog*.**cnt**

where *prog* is the name of the profiled program.

Step 3: Examine the data by running a profiler with the **prof** or **lprof** command.

Each of the following three sections explains one of these steps in detail. Together, they provide an example of how to perform dynamic analysis of a file called **travel.c**.

Creating a Profiled Version of a Program

What must you do to profile a file with **lprof**? Suppose you have a file called **travel.c**. (This is a hypothetical example; CPPT does not include such a file.) Start by creating an executable file (**a.out**) from the source file (**travel.c**). Use the **-ql** option with the **cc** command so that line count data will be saved.

   $ cc -ql travel.c

If you want to use a **cc -c** command line, you must specify **-ql** when you link as well as when you compile.

   $ cc -ql -c travel.c
   $ cc -ql -c misc.c
   $ cc -ql -o travel travel.o misc.o

These sample command lines illustrate what you must do to profile an entire program. However, you may be interested in profiling only a piece of a large program. To profile an individual source file, create a profiled version in the same way: specify the **-ql** option with the **cc** command both when you compile and when you link the files.

For example, suppose you have a program composed of two source files: **travel.c** and **misc.c**. By running **prof** on both files, you find out that 70% of the total execution time can be accounted for by one function in **travel.c**. You now want to examine that function with **lprof** to determine how you can improve its performance. Run the **cc** command with the **-ql** option on the **travel.c** file alone and again when you link **travel.c** and **misc.c**.

   $ cc -ql -c travel.c
   $ cc -c misc.c
   $ cc -ql -o travel travel.o misc.o

The final result will be a program called **travel**.

Running the Profiled Program

Now execute **travel** so that run-time data can be collected. This informa-
tion is stored in a data file called **travel.cnt** in your current directory. When
the program ends, the following message is printed to **stderr**:

```
dumping profiling data from process 'travel' . . .
        CNTFILE 'travel.cnt' created
```

This is how **lprof** handles run-time data by default. However, if you
prefer, you can specify how you want this data to be handled by setting
options for an environment variable called PROFOPTS.

The PROFOPTS Environment Variable

The environment variable PROFOPTS provides run-time control over pro-
filing. When the profiled program is about to terminate, it examines the value
of PROFOPTS to determine how the profiling data is to be handled.

The PROFOPTS environment variable is a comma-separated list of
options interpreted by the program being profiled. If PROFOPTS is not
defined in the environment, then the default action is taken: the profiling
data is saved in a file (with the default name) in the current directory. If
PROFOPTS is set to the null string, no profiling data is produced.

The following options can be specified for PROFOPTS. They are
explained in more detail in the examples.

msg=[y|n]   If **msg=y** is specified, print a message (to **stderr**) stat-
ing that profile data is being created. If **msg=n** is
specified, print only profiling error messages. The
default is **msg=y**.

merge=[y ! n]	If **merge**=**n** is specified, do not merge data files after successive runs; the data file will be overwritten after each execution. If **merge**=**y** is specified, the data will be merged. The merge will fail if the program has been recompiled; the data file will be stored in **TMPDIR**. The default is **merge**=**n**.
pid=[y ! n]	If **pid**=**y** is specified, the name of the data file will include the process ID of the profiled program. This allows the creation of different data files for programs calling **fork**(2). If **pid**=**n** is specified, the default name is used. The default is **pid**=**n**.
dir=*dirname*	Place the data file in the directory *dirname* if this option is specified. Otherwise the data file is created in the directory that is current at the end of execution.
file=*filename*	Use *filename* as the name of the data file in *dir* created by the profiled program if this option is specified. Otherwise the default name is used. (See "Profiling Programs that Fork" for an example.)

Examples of Using PROFOPTS

The following sections provide examples of how PROFOPTS might be used, in typical profiling situations, to tailor the environment for specific tasks.

Turning Off Profiling

If you do not want to profile a particular run, you can set PROFOPTS to the null string on the command line when you run a profiled version of a program.

> $ PROFOPTS=" " travel

However, this value will remain in effect for only one execution of one program.

If you want to turn off profiling for more than one program and/or run, you must export the value of PROFOPTS.

> $ PROFOPTS=" " export PROFOPTS
> $ travel

Exporting the variable eliminates the need to specify it every time you run **travel**. It also makes the value of PROFOPTS applicable to all runs of any profiled programs, not just **travel**. Once you have exported PROFOPTS, it keeps the value you have given it until you unset or redefine that variable.

Merging Data Files

Suppose you are not interested in the data from a single run; you want the information collected from all runs. A data file containing information from multiple executions is called a merged data file. When data files created with the **lprof** compiling option are merged, the execution counts for all files are added together arithmetically.

The following screen shows how you must specify the environment if you want your data files from successive runs to be merged.

```
$ PROFOPTS="merge=y"
$ export PROFOPTS
$ travel


        dumping profiling data from process 'travel' . . .
                CNTFILE 'travel.cnt' created

$ travel


        dumping profiling data from process 'travel' . . .
                CNTFILE 'travel.cnt' updated
```

Keeping Data Files in a Separate Directory

To avoid clutter in your current directory, you may want to create a directory for data files. If you do, be sure to specify that directory on your command line. For example:

> $ PROFOPTS="dir=cntfiles" travel

All the data files will be created in the subdirectory **cntfiles**.

Profiling within a Shell Script

You may want to write a shell script that runs profiled programs automatically. This could be useful for specific tasks that you frequently perform, such as determining coverage. For example:

■ You might not want to receive notification (via a message sent to stderr) that profiling data is being created.

■ You might want to have data merged automatically.

■ You might want to give the data files names that you can associate with a specific test case run.

You can specify these conditions by using PROFOPTS as follows:

> $ PROFOPTS="msg=n, merge=y, file=test1.cnt" myprog < test1

Here, because all the data files in the directory are for the program **myprog**, the file name **test1.cnt** conveys more information than **myprog.cnt**.

Profiling Programs that Fork

If a program uses the system call **fork**(2), the data files of both the parent and child processes will have the same name by default. You can avoid this by using the PROFOPTS option **pid**. By setting **pid**, you ensure that the data file name will include the process ID of the program being profiled. As a result, multiple data files will be created, each with a unique name.

What happens when you run a program that forks without using the **pid** option? If you have set **merge=y**, the data will be merged; data from separate processes will be indistinguishable. If you have set **merge=n**, the last process to dump data will overwrite the data file.

The following screen shows how the **pid** option works. Notice the data files that are created (as reported by the messages sent to stderr) by the command line at the top of the screen.

```
$ PROFOPTS="pid=y" forkprog

    dumping profiling data from process 'forkprog' . . .
            CNTFILE '922.forkprog.cnt' created

    dumping profiling data from process 'forkprog' . . .
            CNTFILE '923.forkprog.cnt' created
```

Interpreting Profiling Output

You can use **lprof** to

■ produce source listing reports of profile data

■ produce summary reports of profile data

■ merge profile data files

Specifying a Program and Data File to lprof

lprof interprets both a profiled program and the data file associated with it to produce profiling information. By default, **lprof** expects the profiled program to be called **a.out**, and the data file, **a.out.cnt**.

To run **lprof** on a program with a name other than **a.out**, specify the name after the **–o** option. For example, to run **lprof** on a program called **sample** use the following command line:

   $ **lprof –o sample**

lprof will assume the data file is called **sample.cnt**.

You can also specify a data file other than **sample.cnt** by using the **–c** option.

   $ **lprof –c newdata.cnt**

The name of the profiled program is stored, exactly as it appears on the command line, in the data file. (Because the **–o** option is not specified, the

profiled program consults the data file to obtain the name of the program.)
Therefore the simplest way of invoking **lprof** is by specifying the name of the
data file and letting **lprof** determine the name of the program. Because the
name of the data file is not stored in the program itself, the reverse is not true:
you cannot specify the name of the program and expect **lprof** to determine the
name of the data file if it is not the default name.

Source Listing Option

Along with profiling information, **lprof** produces a source listing by
default. Once you have executed your profiled program and the data file has
been created, you can view the profile data by entering the following com-
mand:

    $ lprof

lprof output consists of a source listing with profiling information in the left
margin, as shown in the following example:

```
            #include <stdio.h>

            main()
    1 [4]   {
                /* note that declarations are not executable lines
                    and therefore have no line-number or execution
                    status associated with them */

                int i;
    1 [11]     for (i = 0; i < 10; i++)
   10 [12]         sub1();

    1 [14]     }


            sub1()
   10 [17]   {
                /* but here, this declaration is an executable statement */
   10 [19]     int i = 0;

   10 [21]     if (i > 0) {
                    /* next line is an example of code never executed */
    0 [23]         sub2();
                }
                else {
   10 [26]         sub3();
                }
   10 [28]     }


            sub2()
    0 [31]   {
                /* do nothing */
    0 [33]     }


            sub3()
   10 [36]   {
                /* do nothing */
   10 [38]     }
```

Figure 18-18: Example of **lprof** Output

The square brackets enclose line numbers for the file. Each number to the left of a line number shows how many times the corresponding source line was executed.

If you use the **−x** option to **lprof**, the output will highlight the lines that have not been executed. Lines that have been executed will be marked only by line numbers. Lines that have not been executed will be marked with a line number preceded by a [U]. Figure 18-19 shows an example of output produced by the **−x** option.

```
           #include <stdio.h>

           main()
   [4]     {
               /* note that declarations are not executable lines
                   and therefore have no line-number or execution
                   status associated with them */

               int i;

  [11]         for (i = 0; i < 10; i++)
  [12]             sub1();

  [14]     }

           sub1()
  [17]     {
               /* but here, this declaration is an executable statement */
  [19]         int i = 0;

  [21]         if (i > 0) {
                   /* next line is an example of code never executed */
[U] [23]           sub2();
               }
               else {
  [26]             sub3();
               }
  [28]     }

           sub2()
[U] [31]   {
               /* do nothing */
[U] [33]   }

           sub3()
  [36]     {
               /* do nothing */
  [38]     }
```

Figure 18-19: Example of Output Produced by the −x Option

In any **lprof** output, certain lines (such as declarations, comments, and blank lines) do not have line numbers associated with them. This allows you to distinguish between lines that were not executed during a particular run from those that are not executable. In the previous example, neither line 22 nor line 23 in **sub1** was executed, but line 23 is marked with a line number while line 22 is not. This is because line 22 is not executable; line 23 is executable but was not executed in the run that produced this output.

Source Files in a Different Directory

lprof assumes, by default, that the source files for the program you specify are in the current directory. If they are in another directory, you must specify their location with the **–I** option and a path name. For example, to specify source files in the **/usr/src/cmd** directory, use the following command line.

> $ **lprof –o cat –c cat.cnt –I /usr/src/cmd**

In this line **lprof –I** instructs **lprof** to search for the specified source file, **cat.c**, in the specified directory. You can specify multiple **–I** arguments on one command line.

Source Listing for a Subset of Files

If you want profiling output for a limited number of selected files, use the **–r** option with **lprof**.

> $ **lprof –r file1.c –r file2.c**

This command line will produce output only for **file1.c** and **file2.c**. This is useful if you want to examine a few files rather than an entire program.

Summary Option

You can obtain a summary report of the profile data by specifying **lprof –s**.

> $ **lprof –s –c sample.cnt**

Because a source listing is not produced with **lprof –s**, the **–r** and **–I** options do not need to be specified. The following screen shows an example of output produced with the **–s** option.

```
Coverage Data Source: sample.cnt
Date of Coverage Data Source: Mon Apr  7 17:19:43 1986
Object: sample

percent   lines    total    function
covered   covered  lines    name

100.0      4        4       main
 83.3      5        6       sub1
  0.0      0        2       sub2
100.0      2        2       sub3

 78.6     11       14       TOTAL
```

Figure 18-20: Example of **lprof -s** Output

Merging Option

As described in the section on the PROFOPTS environment variable, data files can be merged automatically at run time. You can also merge existing data files with the **lprof** command.

$ **lprof –d** *destfile* **–m** *file1*.**cnt** *file2*.**cnt** *file3*.**cnt**

The command line requires both **–d** and **–m**. The **–m** option takes the names of two or more data files to be merged. The **–d** option specifies the destination file (the new file) that will contain the merged data. The data files must have been created by the same profiled program; if they have not, **lprof** will issue an error message.

```
$ lprof -d merged.cnt -m prog1.cnt prog2.cnt

ERROR: 'prog1', 'prog2'
   Object file entry names & timestamps don't match.
*** no merged output ***
```

However, you may have multiple data files, created by the same program, that have different time stamps. This will happen, for example, if you recompile a program. If you want to merge data from runs of different versions of the same program, you can override the time stamp check by specifying **-T** (time stamp override).

| NOTE | You must be extremely cautious when using the **-T** option. If the control flow of the recompiled program has changed, the new merged data file is very likely to be erroneous; **lprof** will produce an incorrect report. |

Cautionary Notes on Using lprof

This section describes solutions for several problems that may arise while you are using **lprof**.

Trouble at Compile Time

On rare occasions, when compiling a file with the profiling option, you may receive a warning that a particular function is not being profiled.

```
$ cc -c -ql -O file.c


>>> BASICHLK WARNING - not profiling function fname: [trouble at line n]
```

The reason may be that you are using the optimizer together with the profiling option. This is usually a permissible combination of options; occasionally the compiler does not accept it.

You may not need to have the function in question profiled. If not, ignore this warning; data will be collected in the data file for all other functions. If you do want data for the function in question, compile your program again with the profiling option but without optimization. The warning should not reappear.

Non-Terminating Programs

If the profiled program does not terminate, no profiling data will be saved. The profiling data is saved at termination by the system call **exit**(2). If **exit**(2) is never called, no profiling data is saved.

Failure of Data to Merge

If a program has been recompiled, a new data file will be created in a temporary directory. The path name of the new file will be printed to stderr.

Specifying Program Names to lprof

When the profiled program is run, the name of that program is stored, exactly as it appears on the command line, in the data file. The simplest way of invoking **lprof** is by specifying the name of the data file and letting **lprof** determine the name of the program. However, because the name of the data file is not stored in the program itself, the reverse is not true: you cannot specify the name of the program and expect **lprof** to determine the name of the data file if that name is not the default name.

lprof will not be able to display data if you do the following two steps in the order shown:

1. use a relative path name on the command line when you run your profiled program

2. run **lprof** from a different directory specifying only the name of the data file (i.e., without specifying the program name)

When you run **lprof** from a directory other than the one in which you have executed your profiled program, and you have used a relative path name when executing the profiled program, you must specify the **-o** option with either the profiled program's full path name or the program's path name relative to your current directory.

An Example of Using a Relative Path Name

For example, say you are working in a directory called **cur.dir**. You have compiled a program called **newprog.c** and gotten the profiled version, **newprog**. Now you execute **newprog**. A data file called **newprog.cnt** is created in your current directory (**cur.dir**). It includes the name of the profiled version you executed, in the form you entered it on the command line: **newprog**. After **newprog** has finished running, you change directory to **$HOME**. Now you want to examine the results of the execution of **newprog**. From **$HOME** you enter the following command line:

   lprof -c cur.dir/newprog.cnt

Because the data file has stored the name of the profiled file as you entered it on the command line (**newprog**), **lprof** now looks for (and fails to find) **newprog** in the current directory (**$HOME**). You will receive an error message:

   cannot access object file 'newprog'

┌─────┐
│NOTE │   The term object file refers to the profiled version of your file.
└─────┘
  │

To make sure that **lprof** can access the profiled file, specify its relative path (from **$HOME**) with the **-o** option, as follows:

   lprof -c cur.dir/newprog.cnt -o cur.dir/newprog

Trouble at the End of Execution

At the end of execution you may see the following error message:

```
dumping profiling data from process 'a.out' . . .
***unable to seek to symbol table
```

Usually this is caused by running a stripped version of a profiled program. Never strip files to be profiled. If necessary, change makefiles so that they do not produce stripped files.

No Data Are Collected

You may get no data after running a profiled program. The program terminates normally, and you receive neither a message about data being saved, nor an error message. This may be caused by one of two problems:

■ You may not have specified −ql at both compile time and link time. If you forget to specify −ql when you link, the profiled program will run but a data file will not be created.

■ The profiled program may include a call to **_exit** that is causing the program to quit without calling **exit**(2), the procedure that saves your profiling data. Replace calls to **_exit** with calls to **exit**(2) in order to save profiling data.

■ The PROFOPTS variable may be set to NULL.

Data File Cannot Be Found

Occasionally, you may not be able to find the data file, despite the fact that the profiled program has terminated normally and you have received a message saying that the data file has been created.

The profiled program creates the data file in the directory in which the program is located when it terminates. If the program changes directories, the data file may be created in a directory different from both the directory from which you executed the program and the directory in which the shell is located when the program terminates.

Use the **dir** option of PROFOPTS to specify exactly where the data file is to be created so you will be able to find it.

Using lprof **with Shared Libraries**

It is recommended that when profiling with **lprof**, you use archived versions of libraries rather than shared versions. If you must profile with a shared library (for example, if an archived version is unavailable), you must specify all necessary options on the **ld**(1) command line at link time. **ld**(1) is documented in the *Programmer's Reference Manual*.

> NOTE
>
> This is necessary only if you are using CPLU Issue 4.

After compiling as usual with the **-ql** option (as described earlier in this section), link by invoking **ld**(1) directly, as follows:

$ **ld** *user_opts* **/lib/pcrt1.o** *files*.**o -lprof -lld -lm -lc -lg /lib/crtn.o**

user_opts are options, such as **-o prog**, that you normally specify on the **cc**(1) command line. **cc**(1) is also in the *Programmer's Reference Manual*.

You must also check any makefiles to make sure the **ld**(1) command is invoked (instead of the **cc**(1) command) and has the appropriate options, as shown in the preceding example command line. See the **ld**(1) manual page for details about options available with **ld**(1).

Profiling Examples

Improving Performance with prof and lprof

The problem of how to improve program efficiency is addressed by Jon Bentley in *Writing Efficient Programs* (Englewood Cliffs: Prentice-Hall, 1982). Bentley observes that

■ a small part of the code usually accounts for a high percentage of the run time

■ programmers have difficulty identifying the most time consuming parts of the code

To solve the second problem, he recommends the use of profilers. The **prof** and **lprof** profilers can help a programmer locate the time consuming parts of a C program.

| NOTE | The **prof** command is available with Issues 3 and 3.1 of CPLU, but not with Issue 4 or later releases. For those using Issue 4 or a later release, **prof** is available in APU. **prof** is also included with C-FP+. |

Specifically, **prof** provides a time profile, that is, a list of the most time consuming functions and the amount of time taken by each. **lprof** provides a list of the lines that are being executed most frequently. Once these potential problem areas have been identified, it is the programmer's job to rewrite those parts of the code so that the program runs more efficiently.

Although either of these profilers can be used singly, they are most efficient if you use them together, as follows. First, profile your program with **prof** to identify the most time consuming functions. Then profile only those functions (rather than the entire program) with **lprof** to determine which lines are being executed most frequently.

This two-step approach takes the guesswork out of determining which lines of code are the most time consuming. Bentley notes that although programmers want to save time by profiling only selected parts of their code instead of whole programs, they seldom select the correct routines to monitor.

He also emphasizes the importance of profiling programs with data that are typical of data the program will encounter in normal use. Most test cases fail to provide profiling data that are representative of typical usage.

In the next section, an example will be described in detail to illustrate how **prof** and **lprof** can be used to improve program performance.

lprof on lprof

During the development of **lprof** it was observed that the process of merging profile data was slow. The profiling data being merged came from two runs of the C compiler, which is a medium-sized program with 284 functions. It took forty cpu seconds (two minutes of real time) to merge the two coverage files.

The first step was to produce a time profile of **lprof** to see which functions were taking the most time. Here is part of the output from **prof**:

```
%Time  Secands  Cumsecs  #Calls   msec/call  Name
 34.8   13.52    13.52    226638    0.0597    fread
 12.1    4.72    18.24    228254    0.0207    memcpy
  9.5    3.69    21.93     40286    0.0918    CAjump
  9.2    3.60     5.52                        _mcount
  7.7    2.99    28.51       284   10.53      CAfind
  7.6    2.94    31.45     42472    0.0692    malloc
  6.3    2.45    33.90      1154    2.123     read
  3.0    1.17    35.07     40475    0.0289    strcmp
  2.8    1.09    36.16     42471    0.0257    free
  2.5    0.96    37.13         2  482.        creat
  1.4    0.55    37.68         1  550.        fputc
  0.8    0.33    38.01      1431    0.231     lseek
  0.4    0.16    38.17      1518    0.105     fwrite
  0.3    0.11    38.28       569    0.19      CAread
```

Figure 18-21: **prof** Output

The two most time-consuming user functions were **CAjump** and **CAfind**. We wondered why **CAjump** was called 40,286 times and why the average time per call for **CAfind** was so high (10.53 milliseconds).

The next step was to run **lprof** on these two functions. Here are the results of running **lprof** on the function **CAfind**:

.

```
            short
            CAfind(filedata, searchfunc)
            struct caFILEDATA *filedata;
            char *searchfunc;
284 [61]    {
                short ret_code,findflag;
                unsigned char fname_size;
                char *name;

284 [66]        CArewind(filedata);  /* rewind file pointers */
284 [67]        findflag = 1;

40754 [69]      while (findflag)
40470 [70]          if ((fread((char *) &fname_size, sizeof(unsigned char),
                         1, filedata->cov_data_ptr)) > 0) {
40470 [72]              name = (char *) malloc(fname_size+1);
40470 [73]              fread(name, (int)fname_size, 1, filedata->cov_data_ptr);
                        /* make null-terminated */
40470 [75]              name[fname_size] = '\0';
40470 [76]              if (strcmp(name,searchfunc) == 0)
                        {/* this is the function, move
                                    ptr back to beginning of
                                    function name           */
284 [80]                    fseek(filedata->cov_data_ptr,
                                -(long)(fname_size+sizeof(unsigned char)),1);
284 [82]                    ret_code = OK;
284 [83]                    findflag = 0;
                        }
                        else /* this is not it, move to next function */
                        {
40186 [87]                  if (fname_size != EOD)
```

Figure 18-22: **lprof** Output for the Function **CAfind**

```
                                               continued

40186 [88]                        if (CAjump(filedata->cov_data_ptr) == EOF_FAIL)
                                  {/* error - end of file found */
    0 [90]                            ret_code = FUNC_FAIL;
    0 [91]                            findflag = 0;
                                  }
                              }
40470 [94]                    free(name);
                          }
                          else
                          { /* end of file before function found */
    0 [98]                    ret_code = FUNC_FAIL;
    0 [99]                    findflag = 0;
                          }
  284 [101]          return(ret_code);
    0 [102]      }
```

CAfind searches the data file for data pertaining to a particular function. A data file consists of a header section, followed by a section for each function and an end-of-data marker at the end of the file. The coverage data (execution count) for each function is recorded alongside the function's name.

Notice that the **while** loop (shown between lines 70 and 94) was executed 40,470 times; for 284 successful searches, there were 40,186 unsuccessful searches. We were getting a low rate of return for computing resources spent. A look at the **while** loop also shows why **fread** was executed so many times: the loop contains two calls to **fread** (see lines 70 and 73 of the **lprof** output). Finally, the **prof** output reports that **CAjump** was called 40,186 times; once for each unsuccessful search.

Our goals were to minimize the number of unsuccessful searches and, if possible, to decrease the number of calls to **fread**, because these are relatively expensive.

The **lprof** algorithm for merging files consists of two steps: traversing the functions in one of the files sequentially, and calling **CAfind** to locate the data for a given function in the other coverage file.

The first thing that happens in **CAfind** is the resetting of the file pointers so that they point to the first function in the file (line 66). Then, because the given function (which was passed to **CAfind** as an argument) has not been found, the next function in the file is examined to see if it is the correct function. If it is, we are finished. If not, we can skip over the data and try the next function. If we have reached the end of the file, there will be no data for that function in the coverage file and we will return with a failure. By itself, **CAfind** looked fine and there didn't seem to be much we could do to improve its performance.

However, by understanding the entire program, we were able to observe that in almost all situations the order of the coverage data in the two files to be merged was identical. This meant that on subsequent calls to **CAfind**, the next function being sought was immediately after the one found on the last call to **CAfind**. The original implementation did not take advantage of the fact that the search was usually sequential. The file pointers were always reset to the beginning of the file before the search began. Because the functions were in sequential order, this meant that each successive search took progressively longer.

We changed the search strategy so that instead of starting at the beginning of the file on each call to **CAfind**, we started at the place in the file where the previous search had ended. This could have been anywhere in the file. Because files being merged are usually identical, the function being sought is almost always the function following the last one found.

The new search strategy required a slightly more complicated algorithm. Whereas the original strategy demanded only that we check for the end of the file, the new strategy required that we both check for the end of the file and keep track of our current location. The need to do both arose from the sequence of events involved in this type of searching.

The new strategy dictated that each iteration of searching begin where the last search ended. **CAfind** was to search until the function being sought was found. If **CAfind** reached the end of the file before finding that function, it had to continue the search between the first line of the file and the place where it had started the search. Thus **CAfind** had to keep track of when the end of the file was reached. Because the goal of the new strategy was to start each search iteration at the place where the last search had ended, it was obviously necessary to keep track of our current location in the file.

The following screen shows the code for **CAfind** after we changed it to accommodate our new strategy.

```
              short
              CAfind(filedata, searchfunc)
              struct caFILEDATA *filedata;
              char *searchfunc;
284 [61]      {
                  short ret_code;
                  unsigned char fname_size;
                  char *name;
                  long init_loc;

284 [67]          init_loc = -1;
284 [68]          while (1) {
284 [69]              if (init_loc == -1) {
                          /* first time through */
284 [71]                  init_loc = ftell(filedata->cov_data_ptr);
                      }
                      else {
                          /* have we wrapped completely around? */
  0 [75]                  if (ftell(filedata->cov_data_ptr) == init_loc) {
                              /* searched all functions */
  0 [77]                      ret_code = FUNC_FAIL;
  0 [78]                      break;
                          }
                      }
284 [81]              if ((fread((char *) &fname_size, sizeof(unsigned char),
                          1, filedata->cov_data_ptr)) > 0) {
284 [83]                  if (fname_size == EOD) {
                              /* wrap around to beginning */
  0 [85]                      CArewind(filedata);
                              /* go back to top of loop */
                              continue;
                          }
284 [89]                  name = (char *) malloc(fname_size+1);
```

Figure 18-23: **lprof** Output for New Version of Function **CAfind**

```
                                            continued

   284 [90]              fread(name, (int)fname_size, 1,
                         filedata->cov_data_ptr);
                         /* make null-terminated */
   284 [92]              name[fname_size] = '\0';
   284 [93]              if (strcmp(name,searchfunc) == 0)
                         {/* this is the function, move
                                 ptr back to beginning of
                                 function name          */
   284 [97]                 fseek(filedata->cov_data_ptr,
                               -(long)(fname_size+sizeof(unsigned
                               char)),1);
   284 [99]                 ret_code = OK;
                           break;
                         }
                         else /* this is not it, move to next
                                 function */
     0 [104]                 if (CAjump(filedata->cov_data_ptr)
                             == EOF_FAIL)
                             {/* error - end of file found */
     0 [106]                    ret_code = FUNC_FAIL;
                               break;
                             }
                         }
     0 [110]             free(name);
                       }
                       else
                       { /* end of file before function found */
     0 [114]             ret_code = FUNC_FAIL;
                         break;
                       }
                   }

   284 [119]    return(ret_code);
     0 [120]  }
```

Note that not only did we greatly reduce the number of calls to **fread**, but in typical situations we eliminated calls to **CAjump** entirely! Remember,

CAjump originally took 3.69 seconds (9.5% of the total execution time), which was more than any other user function.

The **prof** output for the new version is shown in the following screen.

```
%Time  Seconds  Cumsecs  #Calls   msec/call  Name
25.4    0.54     0.54       298     1.81      read
11.7    0.25     0.79      2002     0.125     malloc
10.6    0.22     1.01      2848     0.079     fread
 8.9    0.19     1.20       579     0.33      lseek
 7.0    0.15     1.35      1518     0.099     fwrite
 6.1    0.13     1.48                         _mcount
 4.2    0.09     1.57       569     0.16      CAread
 3.8    0.08     1.65      4369     0.018     memcpy
 2.8    0.06     1.71       284     0.21      CAor
 2.8    0.06     1.77         2    30.        creat
 2.8    0.06     1.83         1    60.        CAcov_join
 1.9    0.04     1.87       284     0.14      CAfind
 1.9    0.04     1.91       284     0.14      CAdata_entry
 1.9    0.04     1.95      1717     0.023     free
 1.4    0.03     1.98         7     4.        open
```

Figure 18-24: **prof** Output for New Version of **lprof**

The execution time for **CAfind** decreased from 2.99 seconds to 0.04 seconds, and for **CAjump** from 3.69 seconds to 0 seconds. The overall performance for the entire program decreased from forty cpu seconds (two minutes of real time) to two cpu seconds (six seconds of real time).

Improving Test Coverage with lprof

It is difficult to write test suites that fully exercise (cover) programs if you have no way of determining how much of the code is exercised. **lprof** removes the guesswork by showing, on a line-by-line basis, which lines of code are executed. This allows the tester to know exactly what has been tested. It also makes it easier to refine and improve tests.

Suppose we want to measure how well a given test suite tests a program. First we compile the program with **-ql** so that profiling information will be saved. Then we run the program with the tests to get the profiling data. By looking at the summary output, we can see how much of the code is exercised.

```
Coverage Data Source: test.cnt
Date of Coverage Data Source: Wed Mar  5 11:11:58 1986
Object: myprog

percent   lines   total function
covered   covered lines name

 91.5      97      106  compile
100.0      18       18  step
100.0      73       73  advance
100.0       4        4  getrnge
 42.9      12       28  main
100.0      29       29  execute
100.0      19       19  succeed
 42.9       3        7  putdata
  0.0       0       19  regerr
100.0      21       21  fgetl

 85.2     276      324  TOTAL
```

Figure 18-25: **lprof** Summary Output for a Test Suite

More specifically, we can examine individual functions that do not have 100% coverage to find ways of improving the tests.

The rest of this section consists of three examples that show why certain functions may not have 100% coverage. The first example demonstrates how to uncover an option that is usually missed because it is not documented. Another example shows how to uncover a function that is never called. The third example examines code that is never executed because of an error condition that is difficult to produce. Each section also explains how to resolve the problem of lack of coverage.

Example 1: Searching for Undocumented Options

First, examine the function **main** to see what parts of the code are not executed.

```
while((c=getopt(argc, argv, "blcnsvi")) != EOF)
       [32]              switch(c) {
                         case 'v':
  [U] [34]               vflag++;
                         break;
                         case 'c':
       [37]              cflag++;
                         break;
                         case 'n':
       [40]              nflag++;
                         break;
                         case 'b':
       [43]              bflag++;
                         break;
                         case 's':
       [46]              sflag++;
                         break;
                         case 'l':
       [49]              lflag++;
                         break;
                         case 'i':
       [52]              iflag++;
                         break;
                         case '?':
  [U] [55]               errflg++;
       [56]              }
```

Figure 18-26: Fragment of Output from **lprof -x**

The output shows that the **-v** option was not tested. By checking the documentation you can confirm that **-v** is an undocumented option. To correct this, create a test that exercises the **-v** option and add the **-v** option to the manual page.

Example 2: Functions That Are Never Called

None of the lines in the function **regerr** are executed. To find out why, invoke **cscope** and request a list of the functions that call it. **cscope** reports that no function calls **regerr**. Because **regerr** will never be exercised, delete it from the code.

Example 3: Hard to Produce Error Conditions

Look at the function **putdata**:

```
              void
              putdata(output, data)
              char    *data;
              FILE    *output;
       [9]    {
                     /* check for file system out of space */
       [11]          if (fprintf(output, "%s", data) < 0) {
[U] [12]                     fprintf(stderr, "write error with file
                             '%s'", filename);
[U] [13]                     fclose(output);
[U] [14]                     unlink(newreffile);
[U] [15]                     exit(1);
                     }
       [17]   }
```

Figure 18-27: Output from **lprof -x** for Function **putdata**

───

Because this error is hard to reproduce, it usually does not get tested. However, you can simulate this error by writing your own **fprintf** function that returns a value less than 0. This will cause the error recovery part of the function to get exercised, allowing you to see the following error message:

        write error with file '@%#&HP'

Further inspection reveals that the variable *filename* was never initialized. This oversight caused the error message to be garbled.

Appendix A: Index to Utilities

Throughout the text of this guide, commands are discussed without identifying the package to which the command belongs. The assumption has been that all command packages are present on the machine on which you are working.

If some commands seem to produce only a not found message on your computer, it may be that the package to which the command belongs has not been installed. If that happens, check with the administrator of your system.

■ Windowing Utilities

■ Basic Networking Utilities

■ **C Programming Language Utilities**

■ **Advanced C Utilities**

■ **Cartridge Tape Utilities**

■ **Directory and File Management Utilities**

■ Editing Utilities

■ Essential Utilities

■ Graphics Utilities

■ Inter-Process Communications Utilities

■ Line Printer Spooling Utilities

■ **Networking Support Utilities**

■ **Performance Measurement Utilities**

■ **Remote File Sharing Utilities**

■ **Security Administration Utilities**

■ **Software Generation Utilities**

■ Extended Software Generation Utilities

■ Source Code Control System Utilities

■ Spell Utilities

Glossary

Ada   Named after the Countess of Lovelace, the nineteenth century mathematician and computer pioneer, Ada is a high-level general-purpose programming language developed under the sponsorship of the U.S. Department of Defense. Ada was developed to provide consistency among programs originating in different branches of the military. Ada features include packages that make data objects visible only to the modules that need them, task objects that facilitate parallel processing, and an exception-handling mechanism that encourages well-structured error processing.

ANSI standard   ANSI is the acronym for the American National Standards Institute. ANSI establishes guidelines in the computing industry, from the definition of ASCII to the determination of overall datacom system performance. ANSI standards have been established for both the Ada and FORTRAN programming languages, and a standard for C has been proposed.

a.out file   **a.out** is the default file name used by the link editor when it outputs a successfully compiled, executable file. **a.out** contains object files that are combined to create a complete working program. Object file format is described in Chapter 11, "The Common Object File Format," and in **a.out**(4) in the *Programmer's Reference Manual*.

application program   An application program is a working program in a system. Such programs are usually unique to one type of user's work, although some application programs can be used in a variety of business situations. An accounting application, for example, may well be applicable to many different businesses.

archive   An archive file or archive library is a collection of data gathered from several files. Each of the files within an archive is called a member. The command **ar**(1) collects data for use as a library.

argument   An argument is additional information that is passed to a command or a function. On a command line, an argument is a character string or number that follows the command name and is separated from it by a space. There are two types of command-line arguments: options and operands. Options are immediately preceded by a minus sign (–) and change the execution or output of the command. Some options can themselves take arguments. Operands are preceded by a space and specify files or directories that will be operated on by the command. For example, in the command

                                  pr –t –h Heading file

all elements after the **pr** are arguments. **–t** and **–h** are options, **Heading** is an argument to the **–h** option, and **file** is an operand.

For a function, arguments are enclosed within a pair of parentheses immediately following the function name. The number of arguments can be zero or more; if more than two are present, they are separated by commas and the whole list enclosed by the parentheses. The formal definition of a function, such as might be found on a page in Section 3 of the *Programmer's Reference Manual*, describes the number and data type of argument(s) expected by the function.

ASCII   ASCII is an acronym for American Standard Code for Information Interchange, a standard for data representation that is followed in the UNIX System. ASCII code represents alphanumeric characters as binary numbers. The code includes 128 upper- and lower-case letters, numerals, and special characters. Each alphanumeric and special character has an ASCII code (binary) equivalent that is one byte long.

assembler	The assembler is a translating program that accepts instructions written in the assembly language of the computer and translates them into the binary representation of machine instructions. In many cases, the assembly language instructions map 1 to 1 with the binary machine instructions.
assembly language	A programming language that uses the instruction set that applies to a particular computer.
BASIC	BASIC is a high-level conversational programming language that allows a computer to be used much like a complex electronic calculating machine. The name is an acronym for **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode.
branch table	A branch table is an implementation technique for fixing the addresses of text symbols, without forfeiting the ability to update code. Instead of being directly associated with function code, text symbols label jump instructions that transfer control to the real code. Branch table addresses do not change, even when one changes the code of a routine. Jump table is another name for branch table.
buffer	A buffer is a storage space in computer memory where data are stored temporarily into convenient units for system operations. Buffers are often used by programs, such as editors, that access and alter text or data frequently. When you edit a file, a copy of its contents is read into a buffer where you make changes to the text. For the changes to become part of the permanent file, you must write the buffer contents back into the permanent file. This replaces the contents of the file with the contents of the buffer. When you quit the editor, the contents of the buffer are flushed.
byte	A byte is a unit of storage in the computer. On many UNIX Systems, a byte is eight bits (binary digits), the equivalent of one character of text.

byte order

Byte order refers to the order in which data are stored in computer memory.

C

The C programming language is a general-purpose programming language that features economy of expression, control flow, data structures, and a variety of operators. It can be used to perform both high-level and low-level tasks. Although it has been called a system programming language, because it is useful for writing operating systems, it has been used equally effectively to write major numerical, text-processing, and database programs. The C programming language was designed for and implemented on the UNIX System; however, the language is not limited to any one operating system or machine.

C compiler

The C compiler converts C programs into assembly language programs that are eventually translated into object files by the assembler.

C preprocessor

The C preprocessor is a component of the C Compilation System. In C source code, statements preceded with a pound sign (#) are directives to the preprocessor. Command line options of the **cc**(1) command may also be used to control the actions of the preprocessor. The main work of the preprocessor is to perform file inclusions and macro substitution.

CCS

CCS is an abbreviation for C Compilation System, which is a set of programming language utilities used to produce object code from C source code. The major components of a C Compilation System are a C preprocessor, C compiler, assembler, and link editor. The C preprocessor accepts C source code as input, performs any preprocessing required, and passes the processed code to the C compiler. The C compiler produces assembly language code that it passes to the assembler. The assembler, in turn, produces object code that can be linked to other object files by the link editor. The object files produced are in the Common Object File Format (COFF). Other components of CCS include a symbolic debugger, an

	optimizer that makes the code produced as efficient as possible, productivity tools that are used to read and manipulate object files, and libraries that provide run-time support, access to system calls, input/output, string manipulation, mathematical functions, and other code-processing functions.
COBOL	COBOL is an acronym for **CO**mmon **B**usiness **O**riented **L**anguage. COBOL is a high-level programming language designed for business and commercial applications. The English-language statements of COBOL provide a relatively machine-independent method of expressing a business-oriented problem to the computer.
COFF	COFF is an acronym for **C**ommon **O**bject **F**ile **F**ormat. COFF refers to the format of the output file produced on some UNIX Systems by the assembler and the link editor. This format is also used by other operating systems. The following are some of its key features:

☐ Applications may add system-dependent information to the object file without causing access utilities to become obsolete.

☐ Space is provided for symbolic information used by debuggers and other applications.

☐ Users may make some modifications in the object file construction at compile time.

command	A command is the term commonly used to refer to an instruction that a user types at a computer terminal keyboard. It can be the name of a file that contains an executable program or a shell script that can be processed or executed by the computer on request. A command is composed of a word or string of letters and/or special characters that can continue for several (terminal) lines, up to 256 characters. A command name is sometimes used interchangeably with a program name.

command line   A command line is composed of the command name
followed by any argument(s) required by the com-
mand or optionally included by the user. The manual
page for a command includes a command line
synopsis in a notation designed to show the correct
way to type in a command, with or without options
and arguments.

compiler   A compiler transforms the high-level language instruc-
tions in a program (the source code) into object code
or assembly language. Assembly language code may
then be passed to the assembler for further translation
into machine instructions.

core   Core is a (mostly archaic) synonym for primary
memory.

core file   A core file is an image of a terminated process saved
for debugging. A core file is created under the name
"core" in the current directory of the process when
an abnormal event occurs resulting in the process' ter-
mination. A list of these events is found in the **sig-
nal**(2) manual page in Section 2 of the *Programmer's
Reference Manual.*

core image   Core image is a copy of all the segments of a running
or terminated program. The copy may exist in main
storage, in the swap area, or in a core file.

curses   **curses**(3X) is a library of C routines that are designed
to handle input, output, and other operations in
screen management programs. The name **curses**
comes from the cursor optimization that the routines
provide. When a screen management program is run,
cursor optimization minimizes the amount of time a
cursor has to move about a screen to update its con-
tents. The program refers to the **terminfo**(4) database
at run time to obtain the information that it needs
about the screen (terminal) being used. See **ter-
minfo**(4) in the *Programmer's Reference Manual.*

data symbol	A data symbol names a variable that may or may not be initialized. Normally, these variables reside in read/write memory during execution. See text symbol.
database	A database is a bank of information on a particular subject or subjects. On-line databases are designed so that by using subject headings, key words, or key phrases you can search for, analyze, update, and print out data.
debug	Debugging is the process of locating and correcting errors in computer programs.
default	A default is the way a computer will perform a task in the absence of other instructions.
delimiter	A delimiter is an initial character that identifies the next character or character string as a particular kind of argument. Delimiters are typically used for option names on a command line; they identify the associated word as an option (or as a string of several options if the options are bundled). In the UNIX System command syntax, a minus sign (–) is most often the delimiter for option names, for example, –s or –n, although some commands also use a plus sign (+).
directory	A directory is a type of file used to group and organize other files or directories. A directory consists of entries that specify further files (including directories) and constitutes a node of the file system. A subdirectory is a directory that is pointed to by a directory one level above it in the file system organization.
	The ls(1) command is used to list the contents of a directory. When you first log onto the system, you are in your home directory ($HOME). You can move to another directory by using the cd(1) command and you can print the name of the current directory by using the pwd(1) command. You can also create new directories with the mkdir(1) command and remove empty directories with rmdir(1).

A directory name is a string of characters that identifies a directory. It can be a simple directory name, the relative path name or the full path name of a directory.

dynamic linking

Dynamic linking refers to the ability to resolve symbolic references at run time. Systems that use dynamic linking can execute processes without resolving unused references. See static linking.

environment

An environment is a collection of resources used to support a function. In the UNIX System, the shell environment is composed of variables whose values define the way you interact with the system. For example, your environment includes your shell prompt string, specifics for backspace and erase characters, and commands for sending output from your terminal to the computer.

An environment variable is a shell variable such as $HOME (which stands for your login directory) or $PATH (which is a list of directories the shell will search through for executable commands) that is part of your environment. When you log in, the system executes programs that create most of the environmental variables that you need for the commands to work. These variables come from **/etc/profile**, a file that defines a general working environment for all users when they log onto a system. In addition, you can define and set variables in your personal **.profile** file, which you create in your login directory to tailor your own working environment. You can also temporarily set variables at the shell level.

executable file

An executable file is a file that can be processed or executed by the computer without any further translation. That is, when you type in the file name, the commands in the file are executed. An object file that is ready to run (ready to be copied into the address space of a process to run as the code of that process) is an executable file. Files containing shell commands are also executable. A file may be given execute permission by using the **chmod**(1) command. In addition to being ready to run, a file in the UNIX System needs to have execute permission.

exit

Exit is a specific system call that causes the termination of a process. The **exit**(2) call will close any open files and clean up most other information and memory which was used by the process.

exit status: return code

An exit status or return code is a code number returned to the shell when a command is terminated that indicates the cause of termination.

exported symbol

An exported symbol is a symbol that a shared library defines and makes available outside the library. See imported symbol.

expression

An expression is a mathematical or logical symbol or meaningful combination of symbols. See regular expression.

file

A file is an identifiable collection of information that, in the UNIX System, is a member of a file system. A file is known to the UNIX System as an inode plus the information the inode contains that tells whether the file is a plain file, a special file, or a directory. A plain file may contain text, data, programs, or other information that forms a coherent unit. A special file is a hardware device or portion thereof, such as a disk partition. A directory is a type of file that contains the names and inode addresses of other plain, special, or directory files.

file and record locking

The phrase "file and record locking" refers to software that protects records in a data file against the possibility of being changed by two users at the same time. Records (or the entire file) may be locked by one authorized user while changes are made. Other users are thus prevented from working with the same record until the changes are completed.

file descriptor

A file descriptor is a number assigned by the operating system to a file when the file is opened by a process. File descriptors 0, 1, and 2 are reserved; 0 is reserved for standard input (**stdin**), 1 is reserved for standard output (**stdout**), and 2 is reserved for standard error output (**stderr**).

file system

A UNIX System file system is a hierarchical collection of directories and other files that are organized in a tree structure. The base of the structure is the root (/) directory; other directories, all subordinate to the root, are branches. The collection of files can be mounted on a block special file. Each file of a file system appears exactly once in the inode list of the file system and is accessible via a single, unique path from the root directory of the file system.

filter

A filter is a program that reads information from standard input, acts on it in some way, and sends its results to standard output. It is called a filter because it can be used as a data transformer in a pipeline. Filters are different from editors and other commands because filters do not change the contents of a file. Examples of filters are **grep**(1) and **tail**(1), which select and output part of the input; **sort**(1), which sorts the input; and **wc**(1), which counts the number of words, characters, and lines in the input. **sed**(1) and **awk**(1) are also filters but they are called programmable filters or data transformers because, in addition to the data to be transformed, a program must be supplied as input.

flag	A flag or option is used on a command line to signal a specific condition to a command or to request particular processing. UNIX System flags are usually indicated by a leading hyphen (-). The word option is sometimes used interchangeably with flag. Flag is also used as a verb to mean "to point out" or "to draw attention to". See option.
fork	**fork**(2) is a system call that divides a new process into two processes, the parent process and the child processes, with separate, but initially identical, text, data, and stack segments. After duplication, the child (created) process is given a return code of 0 and the parent process is given the process id of the newly created child as the return code.
FORTRAN	FORTRAN is an acronym for **FOR**mula **TRAN**slator. It is a high-level programming language originally designed for scientific and engineering calculations but is now widely adapted for many business uses also.
function	A function is a task done by a computer. In most modern programming languages, programs are made up of functions and procedures which perform small parts of the total job to be done.
header file	A header file is used in programming and in document formatting. In a programming context, a header file is a file that usually contains shared data declarations that are to be copied into source programs as they are compiled. A header file includes symbolic names for constants, macro definitions, external variable references and inclusion of other header files. The name of a header file customarily ends with '.h' (dot-h). Similarly, in a document formatting context, header files contain general formatting macros that describe a common document type and can be used with many different document bodies.
high-level language	A high-level language is a computer programming language such as C, FORTRAN, COBOL, or PASCAL that uses symbols and command statements

representing actions the computer is to perform, the exact steps for a machine to follow. A high-level language must be translated into machine language by a compilation system before a computer can execute it. A characteristic of a high-level language is that each statement usually translates into a series of machine language instructions. Low-level details of the computer's internal organization are left to the compilation system.

host machine

A host machine is the machine on which an **a.out** file is built.

imported symbol

An imported symbol is a symbol used but not defined by a shared library. See exported symbol.

interpreted language

An interpreted language is a high-level language that is not either translated by a compilation system or stored in an executable object file. The statements of a program in an interpreted language are translated each time the program is executed.

Interprocess Communication

Interprocess Communication describes software that enables independent processes running at the same time to exchange information through messages, semaphores, or shared memory.

interrupt

An interrupt is a break in the normal flow of a system or program. Interrupts are initiated by signals that are generated by a hardware condition or a peripheral device indicating that a certain event has happened. When the interrupt is recognized by the hardware, an interrupt handling routine is executed. An interrupt character is a character (normally ASCII) that, when typed on a terminal, causes an interrupt. You can usually interrupt UNIX System programs by pressing the delete or break keys, by typing **CTRL-D**, or by using the **kill**(1) command.

I/O (Input/Output)

I/O is the process by which information enters (input) and leaves (output) the computer system.

kernel

The kernel (comprising 5 to 10 percent of the operating system software) is the basic resident software on which the UNIX System relies. It is responsible for most operating system functions. It schedules and manages work done by the computer and maintains the file system. The kernel has its own text, data, and stack areas.

lexical analysis

Lexical analysis is the process by which a stream of characters (often comprising a source program) is subdivided into its elementary words and symbols (called tokens). The tokens include the reserved words of the language, its identifiers and constants, and special symbols such as =, :=, and ;. Lexical analysis enables you to recognize, for example, that the stream of characters 'print("hello, universe")' is to be analyzed into a series of tokens beginning with the word 'print' [not with the string 'print("h.']. In compilers, a lexical analyzer is often called by the compiler's syntactic analyzer or parser, which determines the statements of the program (that is, the proper arrangements of its tokens).

library

A library is an archive file that contains object code and/or files for programs that perform common tasks. The library provides a common source for object code, thus saving space by providing one copy of the code instead of requiring every program that wants to incorporate the functions in the code to have its own copy. The link editor may select functions and data as needed.

link editor

A link editor, or loader, collects and merges separately compiled object files by linking together object files and the libraries that are referenced into executable load modules. The result is an **a.out** file. Link editing may be done automatically when you use the compilation system to process your programs on the UNIX System. You can also link edit previously compiled files by using the **ld**(1) command.

magic number

The magic number is contained in the header of an **a.out** file. It indicates what the type of the file is, whether it is made up of shared or non-shared text, and on which processor the file is executable.

makefile

A makefile is a file that lists dependencies among the source-code files of a software product and methods for updating them, usually by recompilation. The **make**(1) command uses the makefile to maintain self-consistent software.

manual page

A manual page, or "man page" in UNIX System jargon, is the repository for the detailed description of a command, a system call, a subroutine, or some other UNIX System component.

null pointer

A null pointer is a C pointer with a value of 0.

object code

Object code is executable machine-language code produced from source code or from other object files by an assembler or a compilation system. An object file is a file of object code and associated data. An object file that is ready to run is an executable file.

optimizer

An optimizer, an optional step in the compilation process, improves the efficiency of the assembly language code. The optimizer reduces the space used by, and speeds the execution time of, the code.

option	An option is an argument used in a command line to modify program output by modifying the execution of a command. It is usually one character preceded by a hyphen (-). When you do not specify any options, the command will execute according to its default options. For example, in the command line

        ls -a -l directory

-a and **-l** are the options that modify the **ls**(1) command to list all **directory** entries, including entries whose names begin with a period (.), in the long format (including permissions, size, and date).

parent process	A parent process occurs when a process is split into two, a parent process and a child process, with separate, but initially identical text, data, and stack segments.
parse	To parse is to analyze a sentence in order to identify its components and to determine their grammatical relationship. In computer terminology the word has a similar meaning, but instead of sentences, program statements or commands are analyzed.
PASCAL	PASCAL is a multipurpose high-level programming language often used to teach programming. It is based on the ALGOL programming language and emphasizes structured programming.
path name	A path name is a way of designating the exact location of a file in a file system. It is made up of a series of directory names that proceed down the hierarchical path of the file system. The directory names are separated by a slash character (/). The last name in the path is either a file or another directory. If the path name begins with a slash, it is called a full path name; the initial slash means that the path begins at the **root** directory.

A path name that does not begin with a slash is known as a relative path name, meaning relative to the present working directory. A relative path name may begin either with a directory name or with two dots followed by a slash (../). One that begins with a directory name indicates that the ultimate file or directory is below the present working directory in the hierarchy. One that begins with ../ indicates that the path first proceeds up the hierarchy; ../ is the parent of the present working directory.

permissions

Permissions are a means of defining a right to access a file or directory in the UNIX System file system. They are granted separately to you, the owner of the file or directory, your group, and all others. There are three basic permissions:

☐ Read permission (r) includes permission to cat, pg, lp, and cp a file.

☐ Write permission (w) is the permission to change a file.

☐ Execute permission (x) is the permission to run an executable file.

Permissions can be changed with the **chmod**(1) command (see the *User's/System Administrator's Reference Manual*).

pipe

A pipe causes the output of one command to be used as the input for the next command so that the two run in sequence. You can do this by preceding each command after the first command with the pipe symbol (!), which indicates that the output from the process on the left should be routed to the process on the right. For example, in the command

    who ! wc –l

the output from the **who**(1) command, which lists the users who are logged on to the system, is used as input for the word-count command, **wc**(1), with the **l** option. The result of this pipeline (succession of commands connected by pipes) is the number of people

who are currently logged on to the system.

portability	Portability describes the degree of ease with which a program or a library can be moved or ported from one system to another. Portability is desirable because once a program is developed it is used on many systems. If the program writer must change the program in many different ways before it can be distributed to the other systems, time is wasted, and each modification increases the chances for an error.
preprocessor	Preprocessor is a generic name for a program that prepares an input file for another program. For example, **neqn** and **tbl** are preprocessors for **nroff**. **grap** is a preprocessor for **pic**. **cpp**(1) is a preprocessor for the C compiler.
process	A process is a program that is at some stage of execution. In the UNIX System, it also refers to the execution of a computer environment, including contents of memory, register values, name of the current working directory, status of files, information recorded at login time, etc. Every time you type the name of a file that contains an executable program, you initiate a new process. Shell programs can cause the initiation of many processes because they can contain many command lines.
	The process id is a unique system-wide identification number that identifies an active process. The process status command, **ps**(1), prints the process ids of the processes that belong to you.
program	A program is a sequence of instructions or commands that cause the computer to perform a specific task, for example, changing text, making a calculation, or reporting system status. A subprogram is part of a larger program and can be compiled independently.
regular expression	A regular expression is a string of alphanumeric characters and special characters that describe a character string. It is a shorthand way of describing a pattern to

be searched for in a file. The pattern-matching func-
tions of **ed**(1) and **grep**(1), for example, use regular
expressions. ·

routine
A routine is a discrete section of a program to accom-
plish a set of related tasks

semaphore
In the UNIX System, a semaphore is a sharable short
unsigned integer maintained through a family of sys-
tem calls which include calls for increasing the value
of the semaphore, setting its value, and for blocking
waiting for its value to reach some value. Sema-
phores are part of the UNIX System IPC facility.

shared library
Shared libraries include object modules that may be
shared among several processes at execution time.

shared memory
Shared memory is an IPC (interprocess communica-
tion) facility in which two or more processes can share
the same data space.

shell
The shell is the UNIX System program—**sh**(1)—
responsible for handling all interaction between you
and the system. It is a command language interpreter
that understands your commands and causes the com-
puter to act on them. The shell also establishes the
environment at your terminal. A shell normally is
started for you as part of the login process. Three
shells, the Bourne shell, the Korn shell, and the C
shell, are popular. The shell can also be used as a
programming language to write procedures for a
variety of tasks.

signal: signal number
A signal is a message that you send to processes or
processes send to one another. The most common
signals you might send to a process are ones that
would cause the process to stop: for example, inter-
rupt, quit, or kill. A signal sent by a running process
is usually a sign of an exceptional occurrence that has
caused the process to terminate or divert from the
normal flow of control.

source code	Source code is the programming-language version of a program. Before the computer can execute the program, the source code must be translated to machine language by a compilation system or an interpreter.
standard error	Standard error is an output stream from a program. It is normally used to convey error messages. In the UNIX System, the default case is to associate standard error with the user's terminal.
standard input	Standard input is an input stream to a program. In the UNIX System, the default case is to associate standard input with the user's terminal.
standard output	Standard output is an output stream from a program. In the UNIX System, the default case is to associate standard output with the user's terminal.

stdio: standard input-output

stdio(3S) is a collection of functions for formatted and character-by-character input/output at a higher level than the basic read, write, and open operations.

static linking	Static linking refers to the requirement that symbolic references be resolved before run time. See dynamic linking.
stream	☐ A stream is an open file with buffering provided by the stdio package. ☐ A stream is a full duplex, processing and data transfer path in the kernel. It implements a connection between a driver in kernel space and a process in user space, providing a general character input/output interface for user processes.
string	A string is a contiguous sequence of characters treated as a unit. Strings are normally bounded by white space(s), tab(s), or a character designated as a separator. A string value is a specified group of characters symbolized to the shell by a variable.
strip	**strip**(1) is a command that removes the symbol table and relocation bits from an executable file.

subroutine	A subroutine is a program that defines desired operations and may be used in another program to produce the desired operations. A subroutine can be arranged so that control may be transferred to it from a master routine and so that, at the conclusion of the subroutine, control reverts to the master routine. Such a subroutine is usually called a closed subroutine. A single routine may be simultaneously a subroutine with respect to another routine and a master routine with respect to a third.
symbol table	A symbol table describes information in an object file about the names and functions in that file. The symbol table and relocation bits are used by the link editor and by the debuggers.
symbol value	The value of a symbol, typically its virtual address, is used to resolve references.
syntax	☐ Command syntax is the order in which command names, options, option arguments, and operands are put together to form a command line. The command name is first, followed by options and operands. The order of the options and the operands varies from command to command.
	☐ Language syntax is the set of rules that describes how the elements of a programming language may legally be used.
system call	A system call is a request by an active process for a service performed by the UNIX System kernel, such as I/O, process creation, etc. All system operations are allocated, initiated, monitored, manipulated, and terminated through system calls. System calls allow you to request the operating system to do some work that the program would not normally be able to do. For example, the **getuid**(2) system call allows you to inspect information that is not normally available, since it resides in the operating system's address space.

target machine   A target machine is the machine on which an **a.out** file is run. While it may be the same machine on which the **a.out** file was produced, the term implies that it may be a different machine.

TCP/IP (Transmission Control Protocol/Internetwork Protocol)
      TCP/IP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols that support multi-network applications. It is the Department of Defense standard in packet networks.

terminal definition   A terminal definition is an entry in the **terminfo**(4) database that describes the characteristics of a terminal. See **terminfo**(4) and **curses**(3X) in the *Programmer's Reference Manual*.

terminfo   ☐   **terminfo** is a group of routines within the curses library that handle certain terminal capabilities. For example, if your terminal has programmable function keys, you can use these routines to program the keys.

      ☐   **terminfo** is a database containing the compiled descriptions of many terminals that can be used with **curses**(3X) screen management programs. These descriptions specify the capabilities of a terminal and how it performs various operations (— for example), how many lines and columns it has, and how its control characters are interpreted. A **curses**(3X) program refers to the database at run time to obtain information it needs about the terminal being used.

      See **curses**(3X) in the *Programmer's Reference Manual*. **terminfo**(4) routines can be used in shell programs, as well as C programs.

text symbol   A text symbol is a symbol, usually a function name, that is defined in the **.text** portion of an **a.out** file.

tool   A tool is a program, or package of programs, that performs a given task.

trap   A trap is a condition caused by an error where a process state transition occurs and a signal is sent to the currently running process.

UNIX Operating System

The UNIX Operating System is a general-purpose, multiuser, interactive, time-sharing operating system developed by AT&T. An operating system is the software on the computer under which all other software runs. The UNIX Operating System has two basic parts:

☐ The kernel is the program that is responsible for most operating system functions. It schedules and manages all the work done by the computer and maintains the file system. It is always running and is invisible to users.

☐ The shell is the program responsible for handling all interaction between users and the computer. It includes a powerful command language called shell language.

The utility programs or UNIX System commands are executed using the shell, and allow users to communicate with each other, edit and manipulate files, and write and execute programs in several programming languages.

userid   A userid is an integer value, usually associated with a login name, used by the system to identify owners of files and directories. The userid of a process becomes the owner of files created by the process and descendent (forked) processes.

utility   A utility is a standard, permanently available program used to perform routine functions or to assist a programmer in the diagnosis of hardware and software errors, for example, a loader, editor, debugging, or diagnostics package.

variable	☐ A variable in a computer program is an object whose value may change during the execution of the program, or from one execution to the next.
	☐ A variable in the shell is a name representing a string of characters (a string value).
	☐ A variable normally set only on a command line is called a parameter (positional parameter and keyword parameter).
	☐ A variable may be simply a name to which the user (user-defined variable) or the shell itself may assign string values.
white space	White space is one or more spaces, tabs, or newline characters. White space is normally used to separate strings of characters and is required to separate the command from its arguments on a command line.
window	A window is a screen within your terminal screen that is set off from the rest of the screen. If you have two windows on your screen, they are independent of each other and the rest of the screen.
	The most common way to create windows on a UNIX System is by using the layers capability of the TELETYPE 5620 Dot-Mapped Display. Each window you create with this program has a separate shell running it. Each one of these shells is called a layer.
	If you do not have this facility, the **shl**(1) command, which stands for shell layer, offers a function similar to the layers program. You cannot create windows using **shl**(1), but you can start different shells that are independent of each other. Each of the shells you create with **shl**(1) is called a layer.
word	A word is a unit of storage in a computer that is composed of bytes of information. The number of bytes in a word depends on the computer you are using. The 80286 Computer has 16 bits or 2 bytes per word. The 80386 Computer has 32 bits or 4 bytes per word, and 16 bits or 2 bytes per half word.

Index