

• • •
• •
•

SCO® UNIX® Development System

User Interfaces Guide



SCO® UNIX®
Development System
User Interface Guide



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

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

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

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

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

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

Date: 10 December 1991
Document Version: 3.2.4D

Preface **1**

Using this guide	2
ETI routines	3
Documentation conventions	4
Development System documentation set	5
Using the Documentation	6
Commercial books and articles	7

Chapter 1 **ETI** **9**

What is ETI?	9
The ETI libraries	9
Screen update and cursor movement optimization	10
ETI, terminfo, and termcap	10
ETI programming basics	12
ETI programming needs	13
Compiling a program that uses ETI	17
Running a program that uses ETI	19
Debugging a program that uses ETI	20

Chapter 2 **curses** **21**

Compiling and linking curses programs	21
Return values for curses routines	21
The standard screen	22
Using the standard screen	22
Adding a character	22
Adding a string	23
Printing strings, characters, and numbers	23
Reading a character from the keyboard	24
Reading a string from the keyboard	25
Reading strings, characters, and numbers	25
Moving the current position	26
Inserting a character	27
Inserting a line	28

Deleting a character	28
Deleting a line	29
Clearing the screen	29
Clearing a part of the screen	30
Sending signals from the Terminal	30
Character attributes	31
Setting and unsetting character attributes	32
Turning on and off standout mode	33
Color manipulation	33
How the color feature works	33
Color pairs table	35
Using the COLOR_PAIR(n) attribute	36
Changing color definitions	36
Portability guidelines	37
Other macros and routines	37
start_color()	38
init_pair()	38
init_color	39
Controlling the terminal	40
Echoing characters to the screen	42
Using cbreak mode for input	42
Setting newline mode	42
Setting raw mode	43
Moving the terminal's cursor	43
Saving and restoring the terminal flags	43
Setting a terminal type	44
Reading the terminal name	44
Ensuring the correct terminal settings	45

Chapter 3

curses windows

47

Windows, subwindows, pads, and subpads	47
Creating and using windows	48
Creating a window	48
Creating a subwindow	49
Writing from, and reading to, windows	50
Accessing window structure	51
Overlaying windows	51
Overwriting a screen	52

Moving a window	53
Touching a window	53
Drawing a box	54
Getting the current position	54
Setting window flags	55
Scrolling a window	56
A more efficient refresh	56
Creating and using pads	61
Creating a pad	61
Creating a subpad	61
Refreshing a pad	62

Chapter 4

panel

63

Compiling and linking panel programs	63
Creating panels	63
Elementary panel window operations	64
Fetching pointers to panel windows	64
Changing panel windows	65
Moving panel windows on the screen	66
Moving panels to the top or bottom of the deck	66
Updating panels on the screen	67
Making panels invisible	68
Hiding panels	68
Checking if panels are hidden	69
Reinstating panels	70
Fetching panels above or below given panels	70
Setting and fetching the panel user pointer	71
Deleting panels	74

Chapter 5

menu

75

Compiling and linking menu programs	75
A sample menu	76
Writing menu programs in ETI	76
Some important menu terminology	76
What a menu application program does	77

A sample menu program	78
Creating and freeing menu items	80
Two kinds of menus: single- and multi-valued	82
Manipulating an item's select value in a multi-valued menu	82
Manipulating item attributes	83
Fetching item names and descriptions	83
Setting item options	84
Checking an item's visibility	85
Changing the current default values for item attributes	86
Setting the item user pointer	86
Creating and freeing menus	88
Manipulating menu attributes	90
Fetching and changing menu items	90
Counting the number of menu items	91
Changing the current default values for menu attributes	91
Displaying menus	92
Determining the dimensions of menus	92
Associating windows and subwindows with menus	99
Fetching and changing a menu's display attributes	102
Posting and unposting menus	104
Menu driver processing	106
Defining the key virtualization correspondence	107
ETI menu requests	108
Application-defined commands	111
Calling the menu driver	111
Establishing item and menu initialization and termination routines	118
Fetching and changing the current item	122
Fetching and changing the top row	123
Positioning the menu cursor	125
Changing and fetching the pattern buffer	126
Manipulating the menu user pointer	127
Setting and fetching menu options	129

Chapter 6

form

133

Compiling and linking form programs	133
Writing form programs in ETI	134
Some important form terminology	134
What a typical form application program does	135

A sample form application program	135
Creating and freeing fields	139
Manipulating field attributes	141
Obtaining field size and location information	141
Moving a field	142
Changing the current default values for field attributes	143
Setting the field type to ensure validation	144
Justifying data in a field	149
Setting the field foreground, background, and pad character	151
Some helpful features of fields	152
Setting and reading field buffers	152
Setting and reading the field status	154
Setting and fetching the field user pointer	156
Manipulating field options	158
Creating and freeing forms	161
Manipulating form attributes	164
Changing and fetching the fields on an existing form	164
Counting the number of fields	165
Changing ETI form default attributes	166
Displaying forms	166
Determining the dimensions of forms	166
Associating windows and subwindows with a form	168
Posting and unposting forms	170
Form driver processing	172
Defining the virtual key mapping	173
ETI form requests	175
Application-defined commands	180
Calling the form driver	180
Establishing field and form initialization and termination routines	190
Manipulating the current field	194
Changing the form page	195
Positioning the form cursor	196
Setting and fetching the form user pointer	198
Setting and fetching form options	200
Creating and manipulating programmer-defined field types	202
Building a field type from two other field types	202
Creating a field type with validation functions	203
Freeing programmer-defined field types	206
Supporting programmer-defined field types	207

Chapter 7

Other ETI routines

215

Routines for drawing lines and other graphics	215
Routines for using soft labels	216
Working with more than one terminal	218

Chapter 8

terminfo

221

What every terminfo program needs	222
Compiling and running a terminfo program	222
An example terminfo program	223
Working with the terminfo database	225
Writing terminal descriptions	225
Name the terminal	226
Learn about the capabilities	227
Compile the description	232
Test the description	233
Comparing or printing terminfo descriptions	234
Converting a termcap description to a terminfo description	235

Chapter 9

Program examples

237

The editor program	237
The highlight program	243
The scatter program	244
The show program	246
The two program	248
The window program	251
The colors program	252

Preface

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.

The *User Interfaces Guide* explains **curses** and the Extended Terminal Interface (ETI) package, which can be used to write screen management programs on a SCO UNIX System V/386 system. The ETI package includes libraries for creating panels, menus, and forms, in addition to the **curses** library.

The *User Interfaces Guide* also describes the **terminfo** database and the information necessary to create additional entries.

To start you writing screen management programs as soon as possible, the information in this book does not cover every routine in the ETI libraries. Although it covers all routines in the **panel**, **menu**, and **form** libraries, it covers only the most frequently used routines in the **curses** library.

For more information, this book points you to the **curses(S)**, **terminfo(F)**, and other manual pages in the *Programmer's Reference*.

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 the *C Library Guide* and the **stdio(S)** manual page of the *Programmer's Reference*). 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.

Using this guide

This book contains nine chapters:

Chapter 1, “ETI,” describes the routines and other components that every ETI program needs to work properly, tells you how to compile and run ETI programs, and introduces important concepts such as refreshing.

Chapter 2, “curses,” describes the routines in the `curses` library for writing to, and reading from, a screen and manipulating colors. It also covers the suite of video attributes and options that enable you to enhance your displays with striking visual effects.

Chapter 3, “curses windows,” explains the use of windows, subwindows, pads and subpads. It delves more deeply into the refresh operation and covers the functions `wnoutrefresh()` and `doupdate()`.

Chapter 4, “panel,” 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.

Chapter 5, “menu,” 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.

Chapter 6, “form,” 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.

Chapter 7, “Other ETI routines,” 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.

Chapter 8, “terminfo,” 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 chapter also describes the `terminfo` database, related support tools, and their relationship to the `curses` library.

Chapter 9, “Program examples,” includes programs that illustrate uses of `curses` routines.

ETI routines

This guide 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 $m \times n$ 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
- **FIELDTYPE**—a field attribute that determines what kind of data may occupy the field

Every ETI function is introduced with a **SYNTAX** section that describes the type of its arguments and return value, if any. The first line of the **SYNTAX** section 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:

SYNTAX

```
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.

Documentation conventions

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

Font conventions

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

Development System documentation set

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

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

The books included with the Development System are described here.

Developer's Overview

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

Programmer's Reference Manual

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

Encyclopedia

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

Developer's Topics

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

Programming Tools Guide

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

Debugging Tools Guide

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

User Interfaces Guide

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

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

Device Driver Writer's Guide

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

Macro Assembler Writer's Guide

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

Using the Documentation

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

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

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

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

Commercial books and articles

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

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

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

Marc Rochkind, *Advanced UNIX Programming*.

W. Richard Stevens, *UNIX Network Programming*.

Maurice Bach, *Design of the UNIX Operating System*

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

Chapter 1

ETI

What is ETI?

The Extended Terminal Interface (ETI) is a collection of libraries that can be used in a C program to control a terminal's video display. The ETI libraries provide routines that can create and manipulate windows, menus, and forms. Programs written using the ETI libraries are independent of any terminal type. The ETI libraries make use of the terminal description databases (**terminfo** or **termcap**), that store the control codes and escape sequences used by the different types of terminals. Screen updating is optimized when the ETI libraries are used, so only modified areas of the screen need to be changed.

The ETI libraries

The ETI libraries, discussed individually in later chapters, contain C functions and macros, many of which resemble routines in the standard C library. ETI consists of the following libraries:

Table 1-1 ETI libraries

Library	Description
curses	routines allowing input and output, cursor manipulation, and the creation of, and interaction with, windows and pads
panel	routines allowing windows to be stacked and moved around within the stack
menu	routines allowing the creation of, and interaction with, menus
form	routines allowing the creation of, and interaction with, forms

Screen update and cursor movement optimization

A major feature of ETI is cursor movement optimization. Cursor movement optimization minimizes the amount a cursor has to move around a screen to update it. Cursor movement optimization can be illustrated using a text editor as an example. Assume that the file being edited contains the following sentence:

ETI is a great package for creating forms and menus.

This could be modified to be:

ETI is the best package for creating forms and menus.

The editor only modifies the area of the screen that contained a great to be the best and all other areas of the screen remain untouched. An unoptimized approach might rewrite the entire screen just to reflect this small change. Cursor movement optimization is also known as output optimization because the amount of data transmitted—the output—is minimized.

Cursor movement optimization takes care of updating the screen for the specific terminal on which an ETI program is run. This means that ETI can do whatever is required to update many different terminal types. It searches the terminal description database to find the correct description for the terminal. More information on the terminal description databases can be found in the section “ETI, terminfo, and termcap.”

There are a number of advantages that cursor movement optimization provides for programs that make use of the ETI libraries.

- Screen updates are quicker because only small portions of the screen are modified during each update.
- Time is saved in having to determine how a program should update the screen, because this is already provided by the ETI libraries.
- The load on the communication lines is reduced because less update information must be transferred along those lines.
- Because the best approach to optimizing screen updating can vary from terminal to terminal, this information is retrieved automatically from the terminal description databases so that the program itself does not need to be concerned with many different updating strategies.

ETI, terminfo, and termcap

terminfo and **termcap** are databases that contain descriptions of all the terminals that can be used with the ETI libraries. The **terminfo** database evolved from **termcap** and is more efficiently organized. It also makes available a larger selection of the ETI library routines. The database that is to be used is

specified when the program is compiled. See the section “Compiling a program that uses ETI” later in this chapter for more information on specifying a database.

Programs written using the ETI libraries reference the terminal description database each time they are invoked. The description for a terminal in the database is comprised of a set of capabilities. A detailed description of capabilities can be found in the chapter “terminfo.” When a program begins execution, these capabilities are read into the corresponding capability variables that have been declared by the included header files. When an ETI library routine is called to interact with a terminal, it retrieves the necessary control codes for that terminal from the capability variables. As a result, the program can be run from any terminal that is described in the terminal description database.

To run the ETI program shown in Example 1-1 using, for example, an ANSI terminal, the program needs to be aware of the number of lines and columns available on the terminal screen. This information is stored as part of the *ansi* description in the **terminfo** database with the following two capabilities:

cols#80, lines#25,

The equivalent capabilities in the **termcap** database are:

co#80:li#25

These capability values are loaded into the variables COLS and LINES when the program begins execution. The inclusion of the header file *curses.h* declares these two variables. When the program calls the **move** routine to change the position of the cursor, the new position is based on the values in the variables COLS and LINES rather than on absolute values. Because these values are set automatically, from the capabilities, the program can be executed on any terminal type without a noticeable difference in what is displayed on the screen.

The terminal type used by the ETI libraries is taken from the TERM environment variable. This can be set either on the command line or in a file that is read automatically during the login process, such as *.profile* or *.cshrc*. If the login shell is either the Bourne shell or the Korn shell, TERM can be set using:

TERM=terminaltype; export TERM

If the login shell is the C shell, TERM can be set using:

setenv TERM terminaltype

terminaltype in the above two examples is the name of the terminal being used, as it is defined in either the **terminfo** or **termcap** databases, depending on which of the two the program has been compiled with. For example, for an ANSI terminal, TERM could be set to “ansi”. If the **terminfo** database is being used, a program using the ETI libraries automatically references the file */usr/lib/terminfo/a/ansi* to find the necessary capability information. If the

termcap database is being used, the program finds the information in the file */etc/termcap*. Detailed information on terminal names can be found on the **term(M)** manual page.

It is possible to specify an alternative database for **terminfo** and **termcap** through two environment variables called **TERMINFO** and **TERMCAP** respectively. This is useful for developing terminal descriptions for new terminals and for modifying existing descriptions without actually changing the databases themselves.

The **TERMINFO** environment variable should be set to the pathname of a file containing a terminal description in the same format as a normal **terminfo** terminal description. This file must also be compiled with the **tic(C)** command. Further information on **terminfo** file formats can be found on the **terminfo(F)** manual page and a detailed description of **terminfo** can be found on the **-terminfo(M)** manual page.

The **TERMCAP** environment variable should be set to the pathname of a file containing terminal descriptions in the same format as those in the file */etc/termcap*. This file does not need to be compiled. Further information on **termcap** file formats can be found on the **termcap(F)** manual page. If additional terminal descriptions are needed, then only one environment variable, corresponding to the terminal description database that the program was compiled to use, is required.

Both databases can also be accessed directly through a set of lower-level routines that do not involve the ETI libraries. These can be used to write additional routines that make use of the terminal independence provided by the terminal description databases. In most cases, this is discouraged because of the already extensive screen management capabilities provided by the ETI libraries. Information on these lower-level routines can be found on the **-terminfo(S)** and **termcap(S)** manual pages. These are also discussed further in the chapter entitled “**terminfo**.”

ETI programming basics

Programs written using ETI can make use of one or more of the ETI libraries. Certain routines, however, are required by all such programs regardless of which ETI libraries are used. In addition, the correct libraries must be linked to the program when it is compiled. This can also include the TAM transition library if the program was originally written using the Terminal Access Method (TAM) for use on the AT&T UNIX PC. Once the program has been compiled correctly, it can be executed and, if necessary, debugged. The environment must also be set up correctly so that the compiled program can find the terminal description database.

ETI programming needs

Each program that makes use of the ETI libraries must include the header files for those libraries. Each program must also initialize the data elements that the ETI libraries use and must reset the terminal to a normally usable state when it terminates. To optimize screen updating, all ETI library routines that are called to interact with the terminal do not reflect on the screen automatically. When the screen is ready to be updated, it must be refreshed by calling either the **refresh** or **wrefresh** library routines. These basic elements of ETI are illustrated in the following sample program:

Example 1-1 A sample ETI program

```

/*
 * bullseye: A sample ETI program
 *
 * To compile: cc bullseye.c -lcurses
 *
 */

#include <curses.h>

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

    move( LINES/2 + 1, COLS/2 - 3 );
    addstr("Eye");
    refresh(); /* send output to the terminal screen */
    sleep(2);

    move( LINES/2 - 3, COLS/2 - 4 );
    addstr("Bulls");
    refresh(); /* send more output to terminal screen */
    sleep(2);

    endwin(); /* restore all terminal settings */
}

```

This sample program simply moves the cursor to two different locations near the center of the screen and writes out the words **Eye** and **Bulls**. Cursor movement is done with the **move** library routine and the words are written out using the **addstr** library routine. These are described in more detail in the chapter entitled “curses.” The rest of the sample program is discussed in the following four sections, which outline the four essential needs for any program that uses the ETI libraries.

The header files

The ETI libraries each have a corresponding header file that must be included if routines from the library are to be used in a program. The header files define various data types and macros, and declare constants and routines that can be used in a program to reference the libraries. The header files for each of the libraries are shown in the following table:

Table 1-2 Library and corresponding header files

Library	Header file
curses	curses.h
panel	panel.h
menu	menu.h
form	form.h

The sample program includes the header file *curses.h*. One result of this inclusion is that the constants `LINES` and `COLS` are defined. As explained in the earlier “ETI, terminfo, and termcap” section, these are given values indicating the number of lines and the number of columns that are available on the current terminal screen. In the sample program, these are used to calculate cursor positions for the `move` routine that place the cursor near the middle of the screen. Because these values are relative to the size of the current terminal, the cursor always appears to be in the same place regardless of the terminal from which the program is run. More information on the various constants that are defined for each of the libraries can be found on the corresponding manual pages.

Inclusion of *curses.h* also defines `refresh()`, which is actually a macro defined as follows:

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

Calling `refresh()` in a program is actually a call to the window refresh routine `wrefresh` asking it to refresh the standard screen, `stdscr`. This is explained further in the “Screen updating with `refresh()` and `wrefresh()`” section later in this chapter. Many of the routines in the ETI libraries are macros combining one or more of the routines that are in the actual libraries. Further information on the individual header files can be found in the chapters that correspond to the libraries.

Initialization with `initscr()`

Before any other ETI library routines can be called, the routine `initscr()` must be called. This should be called only once in a program. In the sample program, this is the first routine inside `main`. It is this routine that determines what the terminal type is by reading the `TERM` environment variable. It also loads the capabilities into their corresponding variables. It is at this point that

the variables `LINES` and `COLS` gain their values. These variables are all mentioned in the “ETI, terminfo, and termcap” section. If an error occurs during the initialization, for example, the description for the terminal on which the program is being run does not exist in the terminal description database, then a message is written to `stderr` and the program exits.

Screen updating with `refresh()` and `wrefresh()`

To optimize all screen updates, the ETI library routines do not update the terminal screen directly. Instead, they modify an internal representation of the screen, which represents a window. Windows are explained in more detail in the chapter entitled “curses windows.” The screen is updated to reflect the accumulated changes made to these internally represented windows only when either of the routines `refresh()` or `wrefresh()` is called in a program. `wrefresh()` is used to update windows other than the default window provided by ETI. `wrefresh()` is explained more thoroughly in the chapter entitled “curses windows.”

A default window called `stdscr` (standard screen), which is the size of the current terminal’s screen, is provided automatically for all programs that use the ETI libraries. Another default window called `curscr` (current screen) keeps track of what is currently displayed on the physical screen. When `refresh()` is called, `stdscr` is compared against `curscr` and the differences are updated on the screen so that the physical screen reflects what is on the `stdscr`. Then `curscr` is also updated to match `stdscr` so that it again reflects what is currently displayed on the physical screen. The following example illustrates what happens when you execute the sample ETI program that prints the words `Eye` and `Bulls` at the center of a terminal screen:

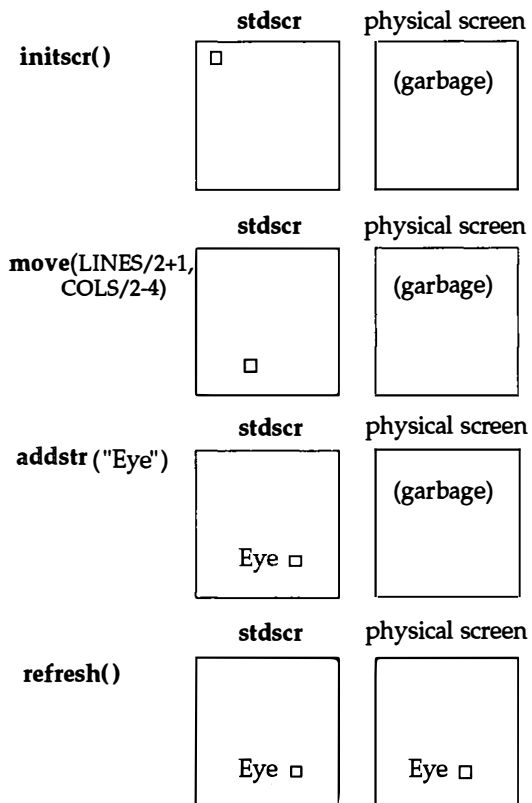


Figure 1-1 Relationship between stdscr and a terminal screen

(Continued on next page.)

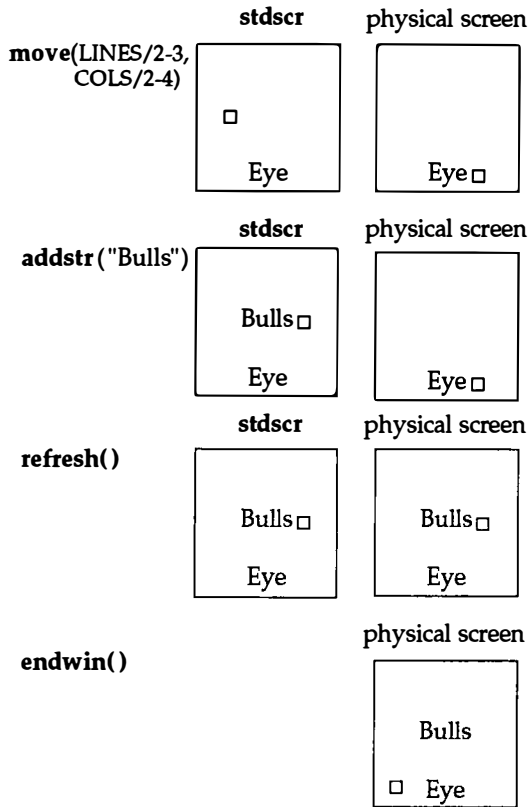


Figure 1-1 Relationship between `stdscr` and a terminal screen (Continued)

Termination with `endwin()`

A program that makes use of the ETI libraries must restore all original terminal settings before the program exits. This is done with the routine `endwin()`. `endwin()` also returns any memory that has been used by the ETI libraries. This means that `stdscr` is no longer accessible. It then positions the cursor in the lower left corner of the screen. In the sample program, `endwin()` is the last routine that is called.

Compiling a program that uses ETI

A program that makes use of the ETI libraries must link in the referenced libraries as the program is being compiled. The necessary libraries are specified to the linker through use of the `-l` option to the `cc(CP)` command. For more information on `cc`, see *Programming Tools Guide*.

The general syntax for compiling a program that uses the ETI libraries is:

```
cc -ooutputfile file1.c file2.c ... [-lX] -lcurses ...
```

-lcurses links in the **curses** library, which must always be included because it contains all of the initialization, basic input and output, refresh, and termination routines. **[-lX]** indicates an optional, additional library, where **X** specifies **panel**, **menu**, **form** or **tam**. If more than one of these additional libraries are used, then each must be preceded by a **-l**. **outputfile** is the name of the output file and **file1.c**, **file2.c** and so on are the names of files that contain the program. The square brackets **[** and **]** are only included to indicate optional libraries. These should not be typed when compiling the program.

NOTE **-lcurses** must appear as the last ETI library, if any of the additional libraries, **panel**, **menu**, or **form**, are used.

The libraries themselves can be found in the directory **/usr/lib**. Each library is named according to the convention **libX.a**, where **X** signifies **curses**, **panel**, **menu**, **form** or **tam**. The **cc(CP)** command understands this naming convention.

A program using the ETI libraries must access one of two terminal description databases called **terminfo** and **termcap**. These are explained further in the section “ETI, terminfo, and termcap.” The default terminal description database is **terminfo**, although this can change. To determine the current default, search the file **/usr/include/curses.h** for these lines:

```
#if !defined( _M_TERMINFO ) && !defined( _M_TERMCAP )
#define _M_TERMINFO
#endif
```

If **_M_TERMINFO** is defined, then the default terminal description database is **terminfo**. If **_M_TERMCAP** is defined, then **termcap** is the default. If the default database is the desired database, then the program can be compiled as illustrated above. If not, then the program can be compiled to use the other database by using the **-D** option to the **cc** command and using the **-l** option to indicate the proper libraries. If the default is **terminfo**, then the **termcap** database can be specified by including the following options when using the **cc** command:

```
cc ... -D_M_TERMCAP ... -ltcap -ltermcap ...
```

If the default is **termcap**, then the **terminfo** database can be specified by including:

```
cc ... -D_M_TERMINFO ... -ltermcap ...
```

Terminal Access Method (TAM) programs

Programs that use the Terminal Access Method (TAM), the screen management package provided on the AT&T UNIX PC, can be compiled to use the ETI libraries by including the TAM transition library. Programs that use TAM should already include the header file *tam.h* at the beginning of each file. If it does not appear, then it can be included at the beginning of each file with the line:

```
#include <tam.h> /* mapping TAM to curses */
```

To compile a program that makes use of the TAM transition libraries, two additions must be made to the `cc` command when the program is compiled. First, the TAM header files are found in the directory */usr/include/tam*, which is not normally read by the `cc` command. This directory can be pointed to with the `-I` option to `cc`. Second, the TAM transition library must be linked with `-ltam` when the program is compiled. A program that makes use of the TAM transition library is compiled using these options to the `cc` command as follows:

```
cc ... -I/usr/include/tam ... -ltam ...
```

Additional information about running a TAM program with the ETI libraries can be found on the **TAM(S)** man page. This includes mappings between the TAM and ETI routines, a description of virtual functions keys, and tips on running a TAM program using the ETI libraries.

Running a program that uses ETI

Programs that use the ETI libraries read environment variables to attain certain information about the terminal from which they are run. The **TERM** environment variable must be set to allow a program using the ETI libraries to determine the terminal type. An explanation of the function of the **TERM** environment variable with examples can be found in the “ETI, terminfo, and termcap” section earlier in this chapter. A program that uses the ETI libraries also makes use of the environment variables **TERMINFO** or **TERMCAP** if they are defined. These are also explained in more detail in the section “ETI, terminfo, and termcap.” Once the required environment variables are set, the program can be executed.

Debugging a program that uses ETI

If a program that uses the ETI libraries does not run as expected, it can be debugged using either **dbxtra** or **Codeview**. Information on these debuggers can be found in the *Debugging Tools Guide*. Both of these debuggers can provide a window that emulates a VT100 terminal. This allows a program using the ETI libraries to be debugged as though it were running on a VT100 terminal. This emulation is provided automatically by **Codeview** when the program is started. In **dbxtra**, this must be set by typing **vt100** at the **(dbxtra)** prompt once **dbxtra** has been started, for example:

```
dbxtra programname  
(dbxtra) vt100
```

Problems that may not be immediately obvious can occur while debugging programs that use the ETI libraries. First, a program that uses the ETI libraries buffers its output to optimize screen updating. The output does not appear on the screen until either one of the **refresh** or **wrefresh** routines has been called. The result of this may be that the program appears to do nothing when what is expected is that some part of the screen be changed. This can be remedied by adding more frequent **refresh** or **wrefresh** calls to the program while it is being debugged, and removing them once the program appears to run correctly.

A second problem that can arise is that breakpoints set on ETI library routines do not work. The likely cause for this is that the routine is actually a macro that references another routine. The solution to this problem is to set the breakpoint on the actual routine and not the macro. For example, a call to **refresh** is really a call to a macro that calls the routine **wrefresh**. To set a breakpoint on **refresh**, the breakpoint should really be set on **wrefresh**. See the "The header files" section earlier in this chapter for more information about macros.

Chapter 2

curses

The **curses** library provides all the basic screen manipulation routines, including input, output, initialization and termination. Not all the routines that are provided in the **curses** library, which number nearly 200, are covered in this chapter. Only the most common are mentioned here. Information on the rest of the routines can be found on the **curses(S)** manual page. A good reference explaining the entire library is *UNIX Curses Explained* by Berny Goodheart.

Compiling and linking curses programs

To use the **curses** routines, specify this line in your C program files:

```
#include <curses.h>
```

Then compile and link them with the command line:

```
cc [ flags ] files -lcurses
```

Return values for curses routines

The header file *curses.h* defines two integer constants that are returned by many of the routines in the **curses** library and subsequently can be used in programs. These are:

OK	returned if a curses routine completes properly
ERR	returned if a curses routine encounters an error

Routines that normally return a pointer return NULL on error. Exceptions to these cases are noted on the **curses(S)** manual page.

The standard screen

When a program that uses the ETI libraries is initialized, it is automatically provided with two default screens, known as windows. The first is called the standard screen, `stdscr`, which represents the entire terminal screen, and the second is called the current screen, `curscr`, which represents the standard screen up until the time of the last call to `refresh`. All `curses` functions that do not specify an alternate window write to the standard screen. Windows are discussed in the chapter “curses windows”. For reasons of optimization, these changes to the standard screen are only updated on the physical terminal screen when the `refresh` function is called. At this time, the standard screen is compared with the current screen and only the differences are updated on the physical terminal screen. The current screen is then updated to be the same as the standard screen.

Using the standard screen

All the basic functions that can be used in a program to allow input from the keyboard and output to the standard screen are provided in the `curses` library. These functions allow reading, writing, and deleting of characters and strings, and also the ability to reposition the cursor. Functions that combine movement with other functions are also included.

Adding a character

The `addch` function adds a given character to the standard screen and moves the character pointer one position to the right. Its argument should be a character declared to be of `chtype`.

SYNTAX

```
int addch(ch)
chtype ch;
```

The constant `OK` is returned if the character is successfully added to the standard screen. `ERR` is returned if an error occurs such as attempting to add a character to the lower right-hand corner of a window that does not scroll.

Writing a character to the standard screen using `addch` works as follows if the current cursor position is (0, 0), then the `addch` function call places the letter `A` at (0, 0) and moves the cursor pointer to (0, 1) on the standard screen:

```
addch('A');
```

Other arguments used by the `addch` function are as follows:

- `\n` Newline. Deletes all characters from the current position to the end of the line and moves the pointer one line down. If the `newline` flag is set, the `addch` function deletes the characters and moves the pointer to the beginning of the next line.

- `\r` **Return.** Moves the pointer to the beginning of the current line.
- `\t` **Tab.** Moves the pointer to the next tab stop, adding enough spaces to fill the gap between the current position and the stop. Tab stops are placed at every eight character positions.

Declaring the argument to be of type **chtype** allows the argument to contain information about attributes as well as the actual character value. More information about attributes can be found in the section “Character attributes” later in this chapter.

Adding a string

The **addstr** function adds a string of characters to the standard screen, placing the first character of the string at the current position and moving the pointer one position to the right for each character in the string. Its argument should be a pointer to a string.

SYNTAX

```
int addstr(str)
char *str;
```

The constant **OK** is returned if the string is successfully added to the standard screen. **ERR** is returned if an error occurs such as attempting to add a string to the lower right-hand corner of a window that does not scroll.

Writing a string to the standard screen works as follows if the current cursor position is (0,0), then the **addstr** function call places the beginning of the string *line* at (0,0) and moves the cursor pointer to (0,4), while adding each character of the string:

```
addstr("line");
```

If the string contains newline, return, or tab characters, the **addstr** function performs the same actions as described for the **addch** function. If the string does not fit on the current line, the string is truncated.

Printing strings, characters, and numbers

The **printw** function prints one or more values to the standard screen, where a value can be a string, a character, or a decimal, octal, or hexadecimal number.

SYNTAX

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

The arguments to `printw` are as follows:

- *fmt* is a pointer to a string that defines the format of the values. A format may be `%s` for a string, `%c` for a character, or `%d`, `%o`, or `%x` for a decimal, octal, or hexadecimal number, respectively. Additional formats can be found on the `printf(S)` manual page. If `%s` is given, the corresponding *arg* must be a character pointer. For other formats, the actual value, or a variable containing the value, can be given.
- *arg* is a value to be printed. If more than one *arg* is given, each must be separated from the preceding argument with a comma (,). For each *arg* given, there must be a corresponding format given in *fmt*.

The constant `OK` is returned if the string is successfully added to the standard screen. `ERR` is returned if an error occurs such as attempting to add a string to the lower right-hand corner of a window that does not scroll.

This function is typically used to copy both numbers and strings to the standard screen at the same time. If, for example, `printf` was used as follows:

```
printw("%s %d", name, 15);
```

If the current cursor position is (0,0), the `printw` function call prints the string given by the variable *name* starting at cursor position (0,0). The `printw` function then prints the number 15 one space after the *name* string.

Reading a character from the keyboard

The `getch` function reads a single character from the terminal keyboard and returns the character as a value.

SYNTAX

```
int getch()
```

The integer value of the character is returned if a character is successfully read. `ERR` is returned if an error occurs such as an attempt to read from the keyboard after input has been redirected.

This function is typically used to read a series of individual characters. For example, in the following program fragment characters are read and stored until a newline character or the end-of-file character is encountered, or until the buffer size has been reached:

```
int c, p[MAX];
int i;

i = 0;
while((c=getch()) != (int)'\n' && i < MAX)
    p[i++] = c;
```

If the terminal is set to ECHO mode, `getch` updates the character to the standard screen; otherwise, the screen remains unchanged. If the terminal is not set to RAW or NOECHO mode, `getch` automatically sets the terminal to CBREAK mode, then restores the previous mode after reading the character. Terminal modes are described in the section “Controlling the terminal” later in this chapter.

Reading a string from the keyboard

The `getstr` function reads a string of characters from the terminal keyboard and updates the string to a given location.

SYNTAX

```
int getstr(str)
char *str
```

str must be a character pointer to the variable or location into which the string is to be stored. When typed at the keyboard, the string must end with a new-line character or with the end-of-file character. The extra character is replaced by a null character when the string is stored. *str* must be declared to have adequate space to store the typed string. The constant `OK` is returned if the string is successfully read. `ERR` is returned if an error occurs such as an attempt to read from the keyboard after input has been redirected.

The `getstr` function is typically used to read lines of text from the keyboard. In the following example `getstr` reads a string from the keyboard and stores it in the array *name*:

```
char name[20];

getstr(name);
```

If the terminal is set to ECHO mode, `getstr` updates the string to the standard screen. If the terminal is not set to RAW or NOECHO mode, the function automatically sets the terminal to CBREAK mode, then restores the previous mode after reading the character. Terminal modes are described in the section “Controlling the terminal” later in this chapter.

Reading strings, characters, and numbers

The `scanw` function reads one or more values from the terminal keyboard and copies the values to given locations. A value can be a string, a character, or a decimal, octal, or hexadecimal number.

SYNTAX

```
int scanw(fmt [,argptr...])
char *fmt
```

The arguments to `scanw` are as follows:

- *fmt* is a pointer to a string defining the format of the values to be read.
- *argptr* is a pointer to the variable to receive a value. If more than one *argptr* is given, each must be separated from the preceding item with a comma (.). For each *argptr* given, there must be a corresponding format given in *fmt*. A format can be *%s* for a string, *%c* for a character, or *%d*, *%o*, or *%x* for a decimal, octal, or hexadecimal number, respectively. Additional formats can be found on the `scanf(S)` manual page.

The constant `OK` is returned if the string is read successfully. `ERR` is returned if the an error occurs during the read, such as a character string in a position where the format string expects an integer.

This function typically reads combinations of characters, strings, and numbers from the keyboard. In the following example, `scanw` reads a name and a number from the keyboard:

```
char name[20];
int id;

scanw("%s %d", name, &id);
```

In this example, the input values are stored in the character array *name* and the integer variable *id*.

If the terminal is set to ECHO mode, the function copies the string to the standard screen. If the terminal is not set to RAW or NOECHO mode, the function automatically sets the terminal to CBREAK mode, then restores the previous mode after reading the character.

Moving the current position

The `move` function moves the cursor to the given position relative to the top left-hand corner of the screen.

SYNTAX

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

The argument *y* indicates the new row position, and the argument *x* indicates the new column position. `move` returns the constant `OK` if the cursor is successfully moved to the new location. `ERR` is returned if the cursor could not be moved to the new location, such as a position outside the dimensions of the screen. The dimensions of the screen are always defined by the values `(0,0)` and `(LINES - 1, COLS - 1)`.

The new cursor position given to the function `move` is always relative to the top left-hand corner of the screen, which is position (0,0), for example:

```
move(5, 4);
```

This would position the cursor at line 5 and column 4.

Combining movement with action

Many operations involve moving the current position of the cursor before the action can be performed. For convenience, a number of functions are provided that combine a read or write function with a movement function performing both actions at once.

SYNTAX

```
mvfunc(y, x, [arg, ...])
```

For all of these type of routines:

- *func* is the name of a function.
- *y* is an integer value that states the line to move to.
- *x* is an integer value that states the column to move to.
- *arg* is the remaining arguments required for the given function.

These functions all return the same values as the equivalent functions without the `mv` prefix.

To illustrate, the `move` and `addch` functions combine to produce `mvaddch`. To move the cursor to the position (10,5) and write the character "X", use:

```
mvaddch(10, 5, 'X');
```

Other functions that combine `move` with another function are described on the `curses(5)` manual page.

Inserting a character

The `<insch` function inserts a character at the current position and shifts the character previously at that position (and all characters to its right) one position to the right.

SYNTAX

```
int insch(ch)
cctype ch;
```

The constant `OK` is returned if the character is successfully added to the standard screen. `ERR` is returned if an error occurs, such as attempting to add a character to the lower right-hand corner of a window that does not scroll.

This function typically inserts a series of characters into an existing line. For example, in the following program fragment, `insch` inserts the number of characters given by `cnt` into the standard screen at the current position:

```
int cnt;
chtype *string;

while(cnt != 0) {
    insch(string[cnt]);
    cnt--;
}
```

Declaring the argument to be of type `chtype` allows the argument to contain information about attributes as well as the actual character value. More information about attributes can be found under the section “Character Attributes” later in this chapter.

Inserting a line

The `insertln` function inserts a blank line at the current position and moves the line previously at that position (and all lines below it) down one line, causing the last line to move off the bottom of the screen.

SYNTAX

```
int insertln()
```

`insertln` takes no arguments. The constant `OK` is returned if the character is successfully added to the standard screen. `ERR` is returned if an error occurs such as attempting to add a character to the lower right-hand corner of a window that does not scroll.

This function inserts additional lines of text in the standard screen. For example, in the following program fragment, `insertln` inserts a blank line when the count in `cnt` is equal to 79:

```
int cnt;

if(cnt == 79) {
    insertln();
}
```

Deleting a character

The `delch` function deletes the character at the current position and shifts the character to the right of the deleted character (and all characters to its right) one position to the left. The last character on the line is replaced by a space.

SYNTAX

```
int delch()
```

delch takes no arguments. The constant **OK** is returned if the character is successfully deleted from the standard screen. **ERR** is returned if an error occurs.

This function typically deletes a series of characters from the standard screen. For example, in the following program fragment, **delch** deletes the character at the current position as long as the count in *cnt* is not zero:

```
int cnt;

while(cnt != 0) {
    delch();
    cnt--;
}
```

Deleting a line

The **deleteln** function deletes the current line and shifts the line below the deleted line (and all lines below it) one line up, leaving the last line on the screen blank.

SYNTAX

```
int deleteln();
```

deleteln takes no arguments. The constant **OK** is returned if the line is successfully deleted from the standard screen. **ERR** is returned if an error occurs.

The **deleteln** function deletes existing lines from the standard screen. For example, in the following program fragment, **deleteln** deletes a line from the standard screen if the count in *cnt* is 79:

```
int cnt;

if(cnt == 79) {
    deleteln();
}
```

Clearing the screen

The **clear** and **erase** functions clear all characters from the standard screen by replacing them with spaces. The functions typically prepare the screen for new text.

SYNTAX

```
int clear()
int erase()
```

Neither **clear** nor **erase** takes an argument. Both always return the constant **OK**.

The **clear** function clears all characters from the standard screen and moves the pointer to (0,0). It then calls **clearok**, which causes the screen to be cleared completely on the next call to **refresh**. This can be used to clear “garbage” on the physical screen that **curses** does not know about. In the following example, **clear** clears the screen if the input value is 12:

```
int c;

if ((c=getch()) == 12) {
    clear();
    refresh();
}
```

The **erase** function clears the standard screen but does not call **clearok**.

Clearing a part of the screen

clrtoebol and **clrtobot** functions clear one or more characters from the standard screen by replacing the characters with spaces. The functions typically prepare a part of the standard screen for new characters.

SYNTAX

```
int clrtoebol()
int clrtobot()
```

Neither **clrtoebol** nor **clrtobot** takes an argument. Both always return the constant **OK**.

The **clrtobot** function clears the screen from the current position to the bottom of the screen. For example, if the current position is (10,0), the **clrtobot** function call clears all characters on line 10 and all lines below line 10. The **clrtoebol** function clears the standard screen from the current position to the end of the current line. For example, if the current position is (10,10), the **clrtoebol** function call clears all characters from (10,10) to (10,79), assuming an 80-character screen; the characters at the beginning of the line remain unchanged. **refresh** must be called after each of the above functions for the changes to appear on the terminal screen. Neither **clrtobot** nor **clrtoebol** changes the current cursor position.

Sending signals from the Terminal

The **flash** and **beep** routines send signals from the terminal. These can be used as warnings or to indicate that a certain action is complete.

SYNTAX

```
int flash()
int beep()
```

Neither returns any useful value.

flash flashes the screen if possible, otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful for catching attention without disturbing others. **beep** rings the terminal bell, otherwise it tries to flash the screen. If for some reason the constant **TRUE** is defined to be something other than the value 1, **beep** does not work.

Character attributes

As mentioned earlier, **addch()** 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 routines described below, the current attributes can be changed. The following is a list of the attributes with meanings:

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
COLOR_PAIR(<i>n</i>)	change foreground and background colors

These attributes are passed to the function **attrset** or any of the other attribute manipulation functions discussed later in this section. These attributes can also be set directly when the characters are passed as arguments to functions such as **addch** by combining the character and the attribute using the bitwise **OR (|)**.

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

The use of attributes can be seen in the following example:

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

Attributes can be turned on separately, such as `attrset(A_BOLD)` in the example, or in combination. To turn on blinking bold text, for example, 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 that may have been made to foreground and background colors.

Occasionally it might be necessary to make a particular portion of text standout noticeably. Because this can vary from terminal to terminal the attribute `A_STANDOUT` can be used to provide this without concern for the particular terminal. `A_STANDOUT` typically translates to reverse video or bold. Two functions, `standout(S)` and `standend(S)` are provided to turn this attribute on and off, respectively. `standend(S)`, in fact, turns off all attributes.

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

Setting and unsetting character attributes

Three functions set and unset the various attributes for text.

SYNTAX

```
int attron(attrs)
cctype attrs;

int attrset(attrs)
cctype attrs;

int attroff(attrs)
cctype attrs;
```

These always return the constant `OK`.

attron turns on the requested attribute **attrs** in addition to any attributes that are currently on. **attrset()** turns on the requested attributes **attrs** in place of any attributes that are currently turned on. **attroff()** turns off the requested attributes **attrs** if they are on. These are used in the **highlight** program in the chapter “Program Examples.”

Turning on and off standout mode

Two functions are provided to turn on and off standout mode.

SYNTAX

```
int standout()  
int standend()
```

These always return the constant OK.

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.

Color manipulation

The color manipulation routines allow colors to be used in the same manner as any other video attribute. The color routines are only supported if the **terminfo** database is used. For more information on **terminfo** see the chapter “ETI”.

This section begins with a description of the color feature at a general level and of the pairs table. 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 programs, 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 and a background color. To use color in a program, the individual colors must first be defined (initialized), color-pairs must be created using those colors, and finally color-pairs must be used as attributes.

A table of defined colors is created automatically when the ETI libraries are initialized. This table has as many entries as the number of colors that the current 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 the intensity of the red, green, and blue light to be specified directly 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 makes 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:

Table 2-1 The default colors table

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

Many color terminals can display only eight colors at the same time. If the terminal can display more than eight, then the table has more than eight entries. The same eight colors will be used to initialize additional entries. If the terminal can display only N colors, where N is less than eight, then only the first N colors shown in the colors table are used.

These color definitions can be changed with the routine `init_color()`, if the terminal is capable of displaying redefined colors. If the terminal is not capable, then the 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.

<code>COLOR_BLACK</code>	0
<code>COLOR_RED</code>	1
<code>COLOR_GREEN</code>	2
<code>COLOR_YELLOW</code>	3
<code>COLOR_BLUE</code>	4
<code>COLOR_MAGENTA</code>	5
<code>COLOR_CYAN</code>	6
<code>COLOR_WHITE</code>	7

Color pairs table

A table of color-pairs is maintained for each program, which has space allocated for as many entries as the number of color-pairs that can be displayed on the terminal screen at the same time. Unlike the colors table, however, there are no default entries in the pairs table. Each color-pair used as an attribute by the program must be initialized using `init_pair`. Each entry in the pairs table has two fields: the foreground color and the background color. For each initialized color-pair, these two fields 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 illustrates how `init_pair()` has been used to initialize color-pair 1 as a blue foreground (entry 4 in the default color table) on a yellow background (entry 3 in the default color table). Color-pair 2 has been initialized as a cyan foreground on a magenta background. Entries in the pairs table that have not been initialized contain zeros corresponding to black on black.

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

Table 2-2 Example of a Pairs Table

Color-pair number	Foreground	Background
0	0	0
1	4	3
2	6	5
3	0	0
4	0	0
5	0	0
.	.	.
.	.	.
.	.	.

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.

NOTE The colors that appear on the screen depend on the color graphics card in the machine. The values placed in the color pairs table may have to be adjusted accordingly.

Using the *COLOR_PAIR(*n*)* attribute

If the default color definitions are chosen, only two things need to be done before using the *COLOR_PAIR(*n*)* attribute. First, the routine *start_color()* must be called. Once this is done, the color-pairs can be initialized 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 a value 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 a color-pair has been initialized, the attribute *COLOR_PAIR(*n*)* can be used in the same way as 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, the routine *attron()* can turn on a color-pair in addition to any other attributes you may currently have turned on:

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

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

*COLOR_PAIR(*n*)* can also be combined with other attributes. For example, this turns on blinking and whatever color-pair 1 had been initialized to be.

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

attron() and *attrset()* are described in the section “Setting and unsetting character attributes” earlier in this chapter.

Changing color definitions

If a terminal is capable of redefining colors, the predefined colors can be changed with the routine *init_color(color, r, g, b)*. The first argument, *color*, is the numeric value of the color that is to be changed. The last three arguments, *r*, *g*, and *b*, are the intensities of the red, green, and blue components, respectively, that the new color will contain. These intensities range in value from 0 to 1000. Once the definition of a color has been changed, all occurrences of that color on your screen change immediately. The definition of color 1 (*COLOR_BLUE* by default) could be changed, for example, 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. However, the capability of terminals vary. For example, if a program is written 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 eight color-pairs.

When writing a program that can be used on different terminals, the following guidelines should be used:

- 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 it has been defined to be, color 0 is always converted to black when used for a background.
- Combine color and other video attributes. Programs that follow this guideline provide some sort of highlighting, even if the terminal is monochrome. On color terminals, as many of the listed attributes as possible are used. On monochrome terminals, only the video attributes are used, and the color attribute is 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 can 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 clear the color-pair field and 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 information about the terminal a 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 information about the colors and color-pairs that are currently defined for a terminal. The routine `color_content()` provides 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 and **OK** otherwise. The routine `pair_content()` permits the colors of a given color-pair to be determined. It returns **ERR** if the color-pair has not been initialized and **OK** otherwise.

The routines `start_color()`, `init_pair()`, and `init_color()` are described in the following sections, with examples of their use. The program `colors` in the chapter “Program examples,” uses the color attribute in windows. The remaining routines in this section are described in more detail on the `curses(S)` manual page.

start_color()

The function `start_color` restores a terminal’s colors to those that were set when the terminal was turned on.

SYNTAX

```
int start_color()
```

It returns **ERR** if the terminal does not support colors, **OK** otherwise. 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` is set to 64.

init_pair()

The routine `init_pair` changes the definition of a color-pair.

SYNTAX

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

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

This is illustrated in the following sample program:

```
#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();
}
```

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

init_color

The function `init_color` changes the definition of a color.

```
SYNTAX

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

It returns `OK` if it was able to change the definition of the color, `ERR` otherwise. 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. This is illustrated in the following sample program:

```
#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();
}
```

When `init_color()` changes 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.

Controlling the terminal

The following sections explain how to set the terminal modes, how to move the cursor, and how to access other aspects of the terminal. These functions should be used only when using other screen-processing functions.

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, **curses** turns off echoing on the UNIX system and does echoing itself. At times, it may be undesirable to have the UNIX system process other characters in the standard way in an interactive screen management program. Some **curses** 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. The following figure shows some of the major routines for controlling input.

Input Options	Characters	
	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 2-1 Input option settings for ETI programs

Every curses program accepting input should set some input options because the default settings cannot be guaranteed to be consistent for all terminals. The combination of **noecho()** and **cbreak()** is most common in interactive screen management programs. **noecho** can prevent characters from being

echoed at the current character position. Instead the program might have them echo at the bottom of the screen instead. When `noecho()` turns off echoing, normal erase and kill processing is still on. The routine `cbreak()` can cause these characters to be uninterpreted.

Do not use the combination of `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. To prevent the terminal from being set incorrectly, routines could be written that contain all the desired settings. Suggestions are provided in the section “Ensuring the correct terminal settings” later in this chapter.

Echoing characters to the screen

The `echo` and `noecho` functions set and unset the ECHO mode for the terminal. When ECHO mode is set, it causes each character typed at the keyboard to be displayed on the terminal screen.

SYNTAX

```
void echo()  
void noecho()
```

When a program begins, ECHO mode is turned on by default.

Using cbreak mode for input

The `cbreak` and `nocbreak` functions set and unset CBREAK mode for reading input. When CBREAK mode is set, a program gets each character as soon as it is typed, except the erase, line kill, and `<Ctrl>D` characters are not interpreted. When CBREAK mode is not set, then input is normally process one line at a time.

SYNTAX

```
void cbreak()  
void nocbreak()
```

When a program begins, CBREAK mode is normally turned on by default.

Setting newline mode

The `nl` and `nonl` functions set and unset NEWLINE mode for a terminal.

SYNTAX

```
void nl()  
void nonl()
```

Setting a terminal to NEWLINE mode causes all newline characters to be mapped to a carriage-return-newline character combination. If NEWLINE

mode is turned off, it causes newline characters to be mapped to themselves. This allows a call to **wrefresh** to do more optimization.

Setting raw mode

The **raw** and **noraw** functions set and unset RAW mode for a terminal.

SYNTAX

```
void raw()  
void noraw()
```

Setting a terminal to RAW mode makes characters available to a program as they are typed. RAW mode also disables the function of the editing and signal keys and disables the mapping of newline characters into carriage-return-newline combinations.

Moving the terminal's cursor

The **mvcur** function moves the terminal's cursor from one position to another in an optimal fashion.

SYNTAX

```
void mvcur(last_y, last_x, new_y, new_x)  
int last_y, last_x, new_y, new_x;
```

The arguments to the function **mvcur** are as follows:

- *last_y* and *last_x* are integer values that give the last line and column positions of the cursor.
- *new_y* and *new_x* are integer values that give the new line and column positions of the cursor.

To move the cursor from the position (10,5) to (3,0) on the terminal screen, **mvcur** is called with the following argument:

```
mvcur(10, 5, 3, 0)
```

The **mvcur** function should be used only in programs that do not use other screen-processing functions. This means the function can perform optimal cursor motion without the aid of the other functions. For programs that do use other functions, the **move**, **wmove**, **refresh**, and **wrefresh** functions must be used to move the cursor.

Saving and restoring the terminal flags

The **savetty** function saves the current terminal flags, and the **resetty** function restores the flags previously saved by the **savetty** function. These functions are performed automatically by the **initscr** and **endwin** functions. They are not required when performing ordinary screen processing.

Setting a terminal type

The **setterm** function sets the terminal type to a given type.

SYNTAX

```
SCREEN *setterm(name)
char *name;
```

The argument *name* is a pointer to a string containing the terminal type identifier. The function is normally called by the **initscr** function, but can be called separately in special cases.

Reading the terminal name

The **longname** function returns the full name of the terminal corresponding to a given **termcap** or **terminfo** identifier.

SYNTAX

```
char* longname(termbuf, name)
char* termbuf;
char* name;
```

The arguments to the function **longname** are as follows:

- *termbuf* is a pointer to the string containing the terminal type identifier.
- *name* is a default string to return if no terminal name is found in *termbuf*.

The terminal type identifier must exist in the */etc/termcap* file or as a file in a subdirectory of */usr/lib/terminfo*.

This function is typically used to get the full name of the terminal currently being used. Note that the current terminal's identifier is stored in the variable *ttytype*, which can be used to receive a new name.

Ensuring the correct terminal settings

All curses programs use a set of functions to initialize and terminate the programs properly, and to set and unset the terminal settings. For convenience, these functions can be placed in routines to ensure that nothing that should be called is forgotten. The following is a set of sample routines that can be used for this purpose:

Example 2-1 Sample routines for ensuring correct terminal settings

```
/*
 * curses initialization and termination routines,
 * and fatal error handler
 *
 * To compile: cc -c file.c
 */

#include < curses.h>

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 ("\n");
    exit (1);
}
```

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.

The routine `start_curses()` calls the required initial settings and sets `CURSES` to `TRUE` to indicate that it has initialized `curses`. The routine `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.

The routine `error` is a universal fatal error handler called whether or not `curses` is initialized. `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()`.

Chapter 3

curses windows

The **curses** library provides routines for creating and manipulating windows, other than the standard screen. The routines that operate on specific windows are generally identical to the routines that operate on the standard screen, except that they take an additional argument, which is a pointer to a specific window. The **curses** library also allows the creation of “pads.” Pads are windows whose size and location are not restricted to the visible part of the terminal screen. Because the window routines are part of the **curses** library, a program creating and using extra windows is compiled in the same way as a program using only the standard screen. See the section “Compiling and linking curses programs” in the chapter “curses.” Only the most common window routines are mentioned here; information on the rest of the routines can be found on the **curses(S)** manual page.

Windows, subwindows, pads, and subpads

Windows can be placed anywhere within the area defined by the dimensions of the standard screen. They can overlap the standard screen or other windows obscuring whatever is beneath them. Windows can also contain subwindows that are associated with them. Any changes made to either a window or a subwindow in the overlapping area affects both of them. Refreshing either updates the changes on the terminal screen. Pads are windows but are not limited to the dimensions of the standard screen. Similarly, subpads are like subwindows but are created within pads. Refreshing pads and subpads differs slightly from refreshing windows. This difference is explained later in this chapter.



The following diagram illustrates how the four types of windows relate to the standard screen:

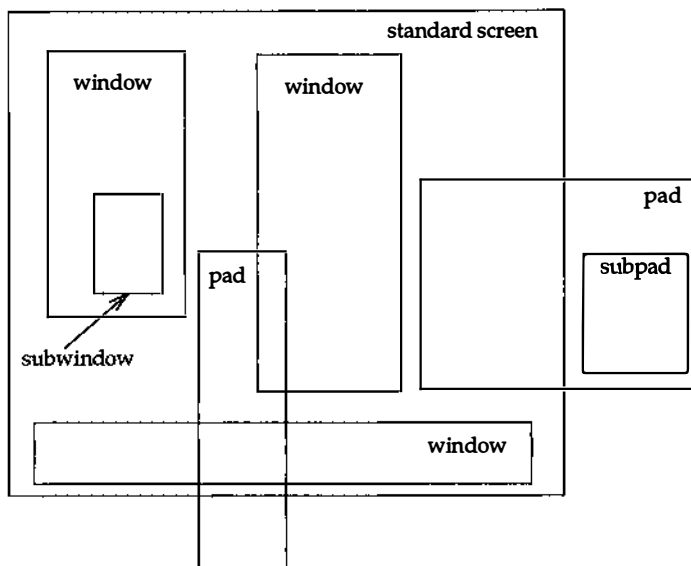


Figure 3-1 Windows, subwindows, pads, and subpads

Creating and using windows

Many of the window routines are identical to the routines used to interact with the standard screen except that they take one additional argument, which is the window to which they apply. One notable difference is that windows must be refreshed independently of the standard screen. This is done with the routine **wrefresh**, which takes the pointer to a window as its single argument. When this is called, only the area of the screen that represents the exposed portion of the window is updated. All the window routines and additional information about those mentioned here can be found on the **curses(S)** manual page.

Creating a window

The **newwin** function creates a window and returns a pointer that can be used in subsequent screen-processing functions.

SYNTAX

```
WINDOW *newwin(lines, cols, begin_y, begin_x)
int lines, cols, begin_y, begin_x;
```

The arguments to **newwin** are as follows:

- *lines* and *cols* are integer values that give the total number of lines and columns in the window, respectively.
- *begin_y* and *begin_x* are integer values that give the line and column positions, respectively, of the upper-left corner of the window when displayed on the terminal screen.

The **newwin** function returns a pointer of type **WINDOW**, which must be assigned to a variable of the same type. The **newwin** function returns the value (WINDOW *) NULL if an error occurs.

The **newwin** function is typically used in programs that maintain a set of windows, displaying different windows at different times or alternating between windows as needed. For example, in the following program fragment, **newwin** creates a new window on the standard screen and assigns a pointer to this window to the variable *midscreen*:

```
WINDOW *midscreen;

midscreen = newwin(5, 10, 9, 35);
```

The window has five lines and 10 columns. The upper-left corner of the window is placed at the position (9,35) on the terminal screen.

If either *lines* or *cols* is zero, the function automatically creates a window that has lines equal to the value **LINES** and columns equal to the value **COLS**, where **LINES** and **COLS** are the predefined constants that give the total number of lines and columns on the terminal screen. For example, the following function call creates a new window whose upper-left corner is at position (0,0) and has lines equal to **LINES** and columns equal to **COLS**:

```
newwin(0, 0, 0, 0)
```

Windows must not be created that exceed the dimensions of the actual screen.

Creating a subwindow

The **subwin** function creates a subwindow and returns a pointer to the new window. A subwindow is a window that shares all or part of the character space of another window and provides an alternate way to access the characters in that space.

SYNTAX

```
WINDOW *subwin(win, lines, cols, begin_y, begin_x)
WINDOW *win;
int lines, cols, begin_y, begin_x;
```



The arguments to **subwin** are as follows:

- *win* is the pointer to the window containing the new subwindow.
- *lines* and *cols* are integer values that give the total number of lines and columns in the subwindow, respectively.
- *begin_y* and *begin_x* are integer values that give the line and column positions, respectively, of the upper-left corner of the subwindow when displayed on the terminal screen.

The **subwin** function returns a pointer of type **WINDOW**, which must be assigned to a variable of the same type. The **subwin** function returns the value (**WINDOW ***) **NULL** if an error occurs.

The **subwin** function typically divides a large window into separate regions. For example, in the following program fragment, **subwin** creates the subwindow named *cmdmenu* in the lower part of the standard screen:

```
WINDOW *cmdmenu;

cmdmenu = subwin(stdscr, 5, 80, 19, 0);
```

In this example, changes to *cmdmenu* affect the standard screen as well.

Writing from, and reading to, windows

The routines that are used to send output to, and get input from, windows are similar to those used with the standard screen. The only difference is that the name of the window must be given as an additional argument. Generally, these functions have names formed by putting the letter **w** at the beginning of the name of the same routine that acts on the standard screen and adding the window name as the first parameter. For example, **addch('c')** becomes **waddch(somewin, 'c')** if you wanted to write the character **c** to the window *somewin*. The following is a list of the window versions of the input and output routines discussed in the chapter “curses”:

- **wmove(win, y, x)**
- **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...])**
- **wgetch(win)**
- **mvwgetch(win, y, x)**
- **wgetstr(win, str)**
- **mvwgetstr(win, y, x, str)**

- **wscanw**(*win*, *fmt* [, *arg...*])
- **mvwscanw**(*win*, *y*, *x*, *fmt* [, *arg...*])
- **winsch**(*win*, *c*)
- **mvwinsch**(*win*, *y*, *x*, *c*)
- **winsertrl**(*win*, *y*, *x*)
- **wdelch**(*win*)
- **mvwdelch**(*win*, *y*, *x*)
- **wdeleteln**(*win*)
- **wclear**(*win*) and **werase**(*win*)
- **wclrtoeol**(*win*) and **wclrtoebot**(*win*)
- **wstandout**(*win*) and **wstandend**(*win*)
- **wattrset**(*win*, *attr*), **wattroff**(*win*, *attr*), and **wattron**(*win*, *attr*)

All the above window routines can be used with pads as well, except for **wrefresh** and **wnoutrefresh**. **wnoutrefresh** is described later in this chapter. Instead of these two routines, the routines **prefresh** and **pnoutrefresh** must be used.

Accessing window structure

The following routines allow you to get useful information on the structure of windows you have created.

getdim (<i>win</i> , <i>y</i> , <i>x</i>)	gets dimensions of window <i>win</i> , specified as coordinates <i>y</i> and <i>x</i>
getorg (<i>win</i> , <i>y</i> , <i>x</i>)	locates origin (upper left point) of <i>win</i> , expressed as <i>y</i> and <i>x</i>
is_standout (<i>win</i>)	determines if window <i>win</i> is in standout mode
iscuroff ()	determines if cursor is in invisible mode
tscroll (<i>win</i> , <i>bf</i>)	enables or disables scrolling in window <i>win</i> based on the value of Boolean flag <i>bf</i> (1 enables, 0 disables)
autoflush (<i>bf</i>)	suspends explicit flushing of output buffers in a refresh () command when the Boolean flag <i>bf</i> is 0. The output buffer is still refreshed when it is full but not on each call to wrefresh (). The default mode is to flush the buffer after each call to wrefresh ().

Overlaying windows

The **overlay** function copies all characters, except spaces, from one window to another, moving characters from their original positions in the first window to identical positions in the second. The function effectively lays the first window over the second, letting characters in the second window that would otherwise be covered by spaces remain unchanged.

SYNTAX

```
int overlay(win1, win2)
WINDOW *win1;
WINDOW *win2;
```

The arguments to the function **overlay** are as follows:

- *win1* is a pointer to the window to be copied.
- *win2* is a pointer to the window to receive the copied text.

The starting positions of *win1* and *win2* must match, otherwise an error occurs. If *win1* is larger than *win2*, the function copies only those lines and columns in *win1* that fit in *win2*.

This function typically builds a composite screen from overlapping windows. For example, in the following program fragment, **overlay** builds the standard screen from two different windows:

```
WINDOW *info, *cmdmenu;

overlay(info, stdscr);
overlay(cmdmenu, stdscr);
refresh();
```

Note that this function cannot be used with disjoint windows. (Windows that do not overlap.)

Overwriting a screen

The **overwrite** function copies all characters, including spaces, from one window to another, moving characters from their positions in the first window to identical positions in the second. The function effectively writes the contents of the first window over the second, destroying the previous contents of the second window. Note that the windows must be overlapping to use this function. **overwrite** typically displays the contents of a temporary window in the middle of a larger window.

SYNTAX

```
int overwrite(win1, win2)
WINDOW *win1;
WINDOW *win2;
```

The arguments to the function **overwrite** are as follows:

- *win1* is a pointer to the window to be copied.
- *win2* is a pointer to the window to receive the copied text.

If *win1* is larger than *win2*, the function copies only those lines and columns in *win1* that fit in *win2*.

For example, in the following program fragment, **overwrite** copies the contents of a window to the standard screen:

```
WINDOW *work;

overwrite(work, stdscr);
refresh();
```

Moving a window

The **mvwin** function moves a given window to a new position on the terminal screen, causing the upper-left corner of the window to occupy a given line and column position.

SYNTAX

```
int mvwin(win, y, x)
WINDOW *win;
int y, x;
```

The arguments to the function **mvwin** are as follows:

- *win* is a pointer to the window to be moved.
- *y* is an integer value giving the line to which the corner is to be moved.
- *x* is an integer value giving the column to which the corner is to be moved.

The function returns **ERR** if it encounters an error, such as an attempt to move part of a window off the edge of the screen.

This function typically moves a temporary window when an existing window under it contains information to be viewed. For example, in the following program fragment, **mvwin** moves the window named *work* to the upper-left corner of the terminal screen:

```
WINDOW *work;

mvwin(work, 0, 0);
```

Touching a window

The **touchwin** function makes the entire contents of a given window appear to be modified, causing a subsequent **refresh** call to copy all characters in the window to the terminal screen.

```
void touchwin(win)
WINDOW *win;
```

win is a pointer to the window to be touched.

This function is typically used when two or more overlapping windows comprise the terminal screen. For example, the following function call touches the window named *leftscreen*:

```
touchwin(leftscreen);
```

A subsequent **refresh** copies all characters in *leftscreen* to the terminal screen.

Drawing a box

The **box** function draws a box around a window using the given characters to form the horizontal and vertical sides.

SYNTAX

```
void box(win, vert, hor)
WINDOW *win;
chtype vert;
chtype hor;
```

The arguments to the function **box** are as follows:

- *win* is a pointer to the desired window.
- *vert* is the vertical character.
- *hor* is the horizontal character.

This function typically distinguishes one window from another when combining windows on a single screen. For example, in the following program fragment, **box** creates a box around the window in the lower half of the screen:

```
WINDOW *cmdmenu;

cmdmenu = subwin(stdscr, 5, 80, 19, 0);
box(cmdmenu, '|', '-');
```

Getting the current position

The **getyx** function copies the current line and column positions of a given window pointer to a corresponding pair of variables.

SYNTAX

```
void getyx(win, y, x)
WINDOW *win;
int y, x;
```

The arguments to the function **getyx** are as follows:

- *win* is a pointer to the window containing the pointer to be examined.
- *y* is the integer variable to receive the line position.
- *x* is the integer variable to receive the column position.

This function typically saves the current position so that the program can return to the position at a later time. For example, in the following program fragment, `getyx` saves the current line and column positions in the variables `line` and `column`:

```
int line, column;

getyx(stdscr, line, column);
```

Setting window flags

The `leaveok`, `scrollok`, and `clearok` functions set or clear the *cursor*, *scroll*, and *clear-screen* flags. The flags control the action of the `refresh` function when it is called for the given window.

The `leaveok` function sets or clears the *cursor* flag, which defines how the `refresh` function places the terminal cursor and the window pointer after updating the screen. If the flag is set, `refresh` leaves the cursor after the last character to be copied and moves the pointer to the corresponding position in the window. If the flag is cleared, `refresh` moves the cursor to the same position on the screen as the current pointer position in the window.

SYNTAX

```
void leaveok(win, state)
WINDOW *win;
bool state;
```

The arguments to the function `leaveok` are as follows:

- *win* is a pointer to the window containing the flag to be set.
- *state* is a Boolean value defining the state of the flag.

If *state* is `TRUE`, the flag is set; if it is `FALSE`, the flag is cleared. For example, the following function call sets the *cursor* flag:

```
leaveok(stdscr, TRUE);
```

The `scrollok` function sets or clears the *scroll* flag for the given window. If the flag is set, scrolling through the window is allowed. If the flag is clear, no scrolling is allowed. The function call has the following form:

SYNTAX

```
void scrollok(win, state)
WINDOW *win;
bool state;
```

The arguments to the function `scrollk` are as follows:

- *win* is a pointer to a window.
- *state* is a Boolean value defining how the flag is to be set.

If *state* is `TRUE`, the flag is set; if it is `FALSE`, the flag is cleared. The flag is initially clear, making scrolling illegal.

The `clearok` function sets and clears the *clear* flag for a given screen. The function call has the following form:

```
void clearok(win, state)
WINDOW *win;
bool state;
```

The arguments to the function `clearok` are as follows:

- *win* is a pointer to the desired screen.
- *state* is a Boolean value.

The function sets the flag if *state* is `TRUE`, and clears the flag if it is `FALSE`. For example, the following function call sets the *clear* flag for the standard screen:

```
clearok(stdscr, TRUE)
```

When the *clear* flag is set, each `refresh` call to the given screen automatically clears the screen by passing a clear-screen sequence to the terminal. This sequence affects the terminal only; it does not change the contents of the screen.

If `clearok` sets the *clear* flag for the current screen *curscr*, each call to `refresh` automatically clears the screen, regardless of which window is given in the call.

Scrolling a window

The `scroll` function scrolls the contents of a given window upward one line.

```
SYNTAX

void scroll(win)
WINDOW *win
```

Here *win* is a pointer to the window to be scrolled.

A more efficient refresh

The `wrefresh` routine updates the contents of a window to the screen. It in turn calls the routines `wnoutrefresh` and `doupdate`. If `wnoutrefresh` and `doupdate` are used directly, the refresh can be done more efficiently than if `wrefresh` was used 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 several windows are to be updated at once, then calling **wrefresh** results in alternating calls to **wnoutrefresh** and **doupdate**, causing several bursts of output to the screen. By calling **wnoutrefresh** for each window and then **doupdate** only once, the total number of characters transmitted and the processor time used can be minimized. The following is a sample program that uses only one **doupdate**:

Example 3-1 A sample program using a more efficient refresh

```
/*
 * A sample program using a more efficient refresh
 *
 * To compile: cc file.c -lcurses
 */

#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();
}
```

The effect of using **wnoutrefresh** and **doupdate** instead of **wrefresh** in Example 3-1 above sample program can best be shown in Figure 3-2

(25)

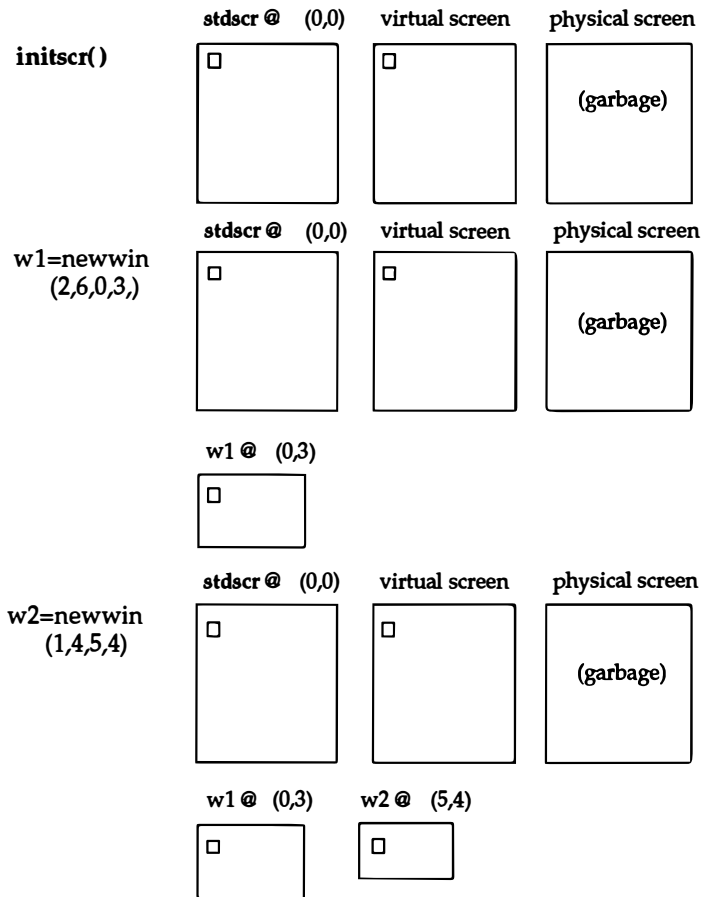


Figure 3-2 Relationship between a window and a terminal screen

(continued)

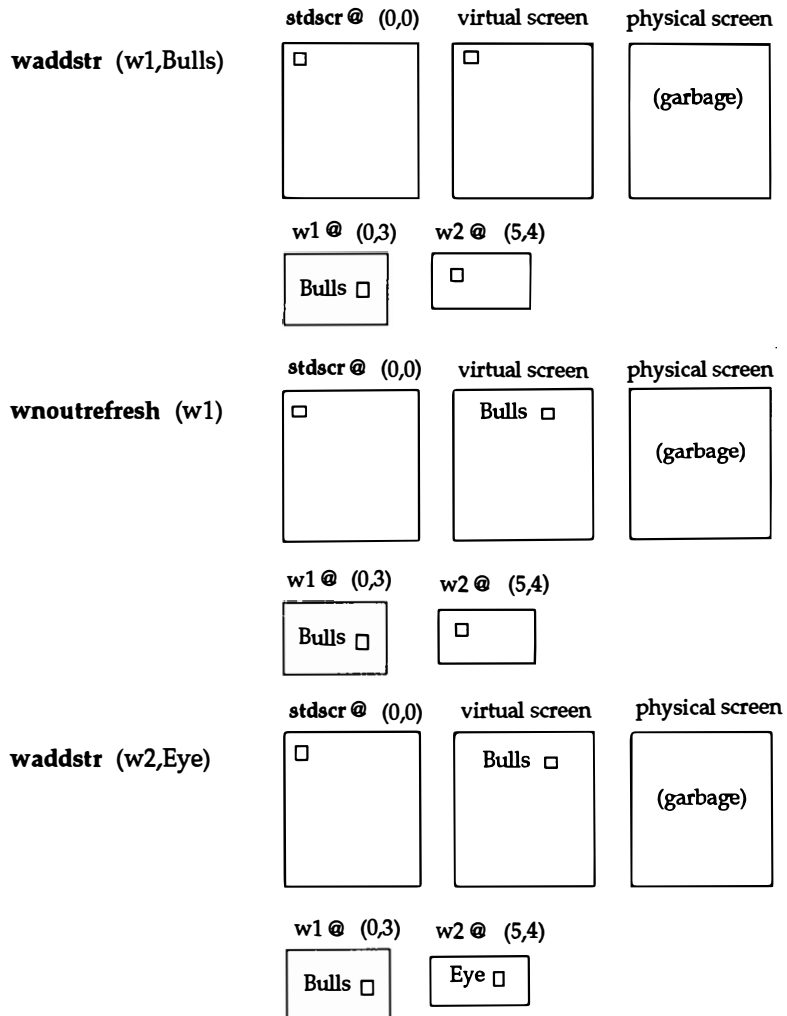


Figure 3-2 Relationship between a window and a terminal screen

(continued)

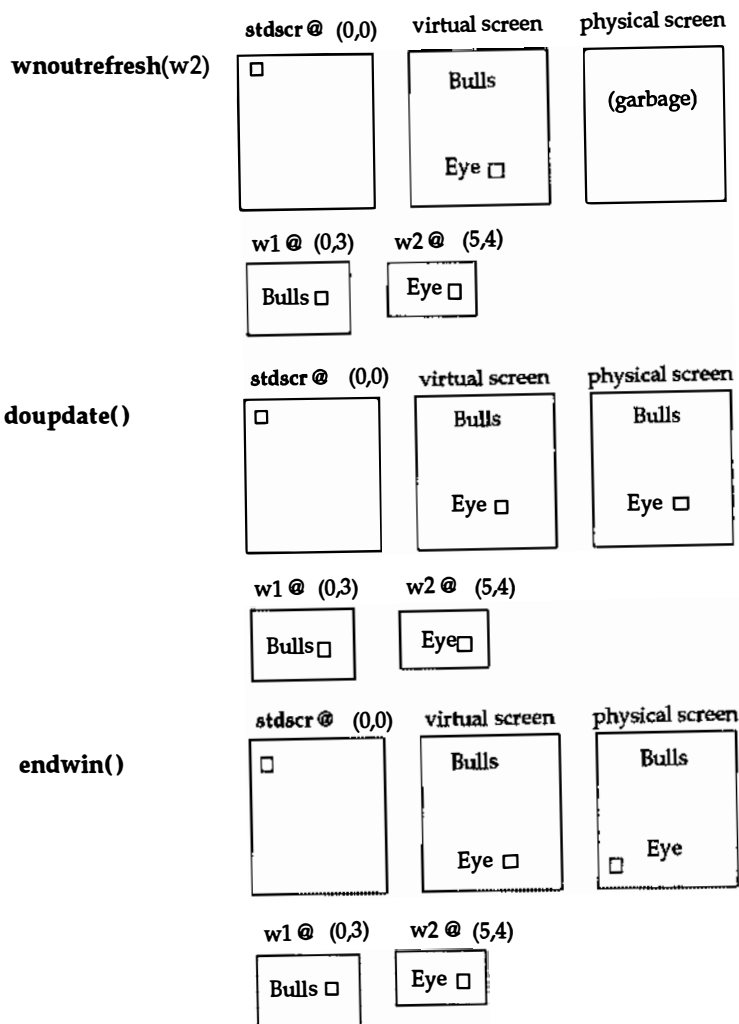


Figure 3-2 Relationship between a window and a terminal screen

Similarly, the routines `pnoutrefresh` and `doupdate` can be used instead of `refresh` as a way to refresh pads more efficiently.

Creating and using pads

Pads are windows that are not restricted in size to the area defined by the physical screen. The amount and portion of a pad that is displayed on the physical screen is determined by the routine used to refresh the pad. All of the window routines that can be used on windows can also be used on pads except that there are separate routines for creating a pad, creating a subpad, and refreshing a pad. These are outlined in the following sections. More information about pads can be found on the **curses(S)** manual page.

Creating a pad

The **newpad** function creates a pad and returns a pointer that can be used in subsequent screen-processing functions in the same way the **newwin** creates a new window. Unlike windows, pads are not associated with any portion of the screen so the location parameters are not required.

SYNTAX

```
WINDOW *newpad(lines, cols)
int lines, cols;
```

lines and *cols* are integer values that give the total number of lines and columns in the pad, respectively.

The **newpad** function returns a pointer of type **WINDOW**, which must be assigned to a variable of the same type. The **newpad** function returns the value (**WINDOW ***) **NULL** if an error occurs.

Because pads are not associated with any particular portion of the screen, the portion of the pad that is displayed is determined when the pad is refreshed. See the section “Refreshing a pad” later in this chapter for more information.

Creating a subpad

The **subpad** function creates a subpad and returns a pointer to the new pad. A subpad is a pad that shares all or part of the character space of another pad and provides an alternate way to access the characters in that space.

SYNTAX

```
WINDOW *subpad(pad, lines, cols, begin_y, begin_x)
WINDOW *pad;
int lines, cols, begin_y, begin_x;
```

The arguments to **subpad** are as follows:

- *pad* is the pointer to the pad to contain the new subpad.
- *lines* and *cols* are integer values that give the total number of lines and columns in the subpad, respectively.
- *begin_y* and *begin_x* are integer values that give the line and column position on the pad that the upper-left corner of the subpad is to be located.

The **subpad** function returns a pointer of type **WINDOW**, which must be assigned to a variable of the same type. The **subpad** function returns the value (**WINDOW ***) **NULL** if an error occurs.

Refreshing a pad

The **prefresh** function refreshes a pad or a subpad. **prefresh** is similar to **wrefresh** but because pads are not normally associated with any portion of the screen, additional parameters are needed to determine which portion of the pad is to be updated to the screen and on which area of the screen it is to be displayed.

SYNTAX

```
int prefresh(pad, prow, pcol, sminrow, smincol, smaxrow, smaxcol)
WINDOW *pad
int prow, pcol;
int sminrow, smincol, smaxrow, smaxcol;
```

The arguments to **prefresh** are as follows:

- *pad* is the pointer to the pad to be refreshed.
- *prow* and *pcol* are integer values that specify the upper left-hand corner of the area within the pad that is put on the screen.
- *sminrow*, *smincol*, *smaxrow*, and *smaxcol* are integer values that specify the rectangular area on the terminal screen that is to be updated.

The routine **pnoutrefresh** can be used in conjunction with the function **doupdate**, instead of **prefresh**, in the same way that **wnoutrefresh** and **doupdate** can be used instead of **wrefresh**. For further information about this alternative, see the section “A more efficient refresh.”

Chapter 4

panel

Recall that a window is a rectangular area of the terminal screen on which you can write using `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 `curses` 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
```

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.

SYNTAX

```
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 **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 **curses** 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 **curses** window associated with it. To retrieve a pointer to this window, use function **panel_window()**.

```
SYNTAX

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 `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.

SYNTAX

```
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 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 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.

SYNTAX

```
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 `curses` 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 `curses` 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.

SYNTAX

```
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 **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.

```
SYNTAX
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 **curses** routines **wnoutrefresh()** and **wrefresh()** when working with panels.

NOTE In general, do not use the **curses** 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 **curses** 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 screen. 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.

SYNTAX

```
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.

SYNTAX

```
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.

SYNTAX

```
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.

SYNTAX

```
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.

SYNTAX

```
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 the following example, create a hidden panel for pop-up messages.

Example 4-1 Example employing a panel user pointer

```
/*
 * message: An example employing a panel user pointer.
 * Displays a pop-up message.
 *
 * To compile: cc message.c -lpanel -lcurses
 *
 */

#include <curses.h>
#include <panel.h>

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

```
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;

    start_curses();
    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 */

    show_mess = TRUE;
    display_deck (show_mess);
    end_curses();
}
```

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 **curses** function `delwin()`.

SYNTAX

```
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);
```

Chapter 5

menu

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, the application program responds accordingly. This response may be to generate a message, display another menu, or take some other action.

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
```

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

A sample menu

Example 5-1 displays a sample menu.

Example 5-1 A sample menu

```

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

Choose an item

```

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 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 curses

A sample menu program

Example 5-2 shows the ETI code necessary for generating the menu of colors in Example 5-1.

Example 5-2 Sample menu program to create a menu in ETI

```

/*
 * menu1: Posts a menu, waits, goes away.
 *
 * To compile: cc menu1.c -lmenu -lcurses
 *
 */

#include <curses.h>
#include <menu.h>

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

ITEM * items[13];

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

    /* 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++);

        /* curses termination */
        endwin ();
        exit (0);
    }

```

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 **curses** 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 **curses** 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 demonstration programs delivered with the ETI product. **curses** 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()**.

SYNTAX

```
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()**. Example 5-3 shows how you might create an item array of the planets of our solar system.

Example 5-3 Creating an array of items

```

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;

```

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()**.

SYNTAX

```

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.

SYNTAX

```
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.

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_REQUEST_DENIED</code>	- 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 Example 5-4 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.

Example 5-4 Using `item_value()` in menu processing

```
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;
    }
}
```

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.

```
SYNTAX

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.

```
SYNTAX

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.

SYNTAX

```
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.

SYNTAX

```
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.

SYNTAX

```
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 *)`. Example 5-5 shows how to use an item’s user pointer with a struct `ITEM_ID`, which stores biological information.

Example 5-5 Using an item user pointer

```

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;

```

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()`.

SYNTAX

```
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 Example 5-2 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 terminator 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()**.

```
SYNTAX

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.

```
SYNTAX

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:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- NULL menu pointer or NULL associated item array
<code>E_POSTED</code>	- menu is posted
<code>E_CONNECTED</code>	- 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 Example 5-6 whose code changes the items associated with a previously created menu.

Example 5-6 Changing the items associated with a 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 */
```

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.

SYNTAX

```
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 previously 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_ROWMAJOR` 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.

SYNTAX

```
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 5-1 and 5-2. 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 5-1 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 5-1 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 5-2. The enclosing block in each case indicates the items displayed at one time.

0	1
2	3

4

Row Major

0	3
1	4

2

Column Major

Maximum Rows 2

Figure 5-1 Examples of menu format (2, 2)

0	1
2	3
4	

Row Major

0	3
1	4
2	

Column Major

Maximum Rows 3

Figure 5-2 Examples of menu format (3, 2)

For a larger example, consider Figure 5-3. 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

Row Major

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

4 10 16

5 11 17

Column Major

Figure 5-3 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:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- rows < 0 or cols < 0
<code>E_POSTED</code>	- 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.

SYNTAX

```
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.

SYNTAX

```
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:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- null menu pointer
<code>E_NOT_CONNECTED</code>	- 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.

SYNTAX .

```
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, (C) the menu window is NULL, which by convention means that ETI uses `stdscr` as the menu window; and (S) 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 5-4.

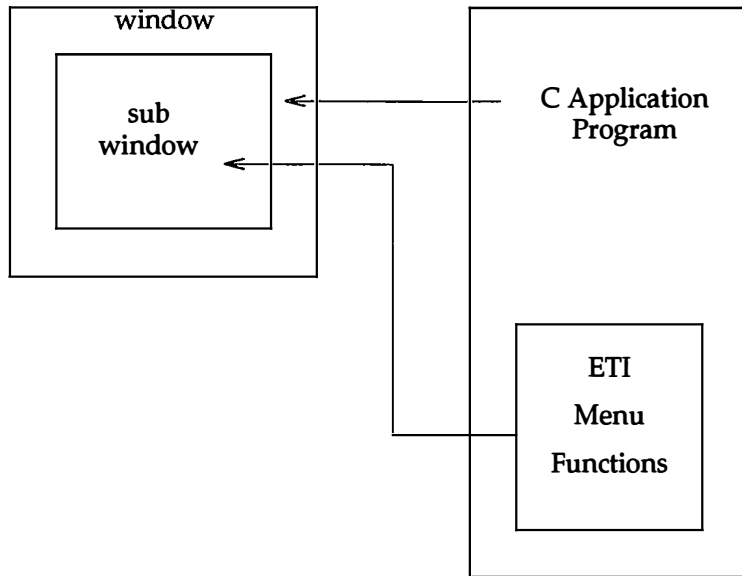


Figure 5-4 Menu functions write to subwindow, application to window

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

Example 5-7 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(S)` manual page.)

Example 5-7 Creating a menu with a border

```

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 */
}

```

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. 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.

```
SYNTAX

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 nonselectable 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.

SYNTAX

```
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.

Example 5-8 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 Example 5-7. Function **erase_menu()** unposts the menu and erases its associated window and subwindow.

Example 5-8 Sample routines displaying and erasing menus

```
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);
}
}
```

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(S)** 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.

SYNTAX

```
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 Example 5-9, 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.

Example 5-9 Sample routine that translates keys into menu requests

```
/* define application commands -- see comments at end */
#define QUIT          (MAX_COMMAND + 1)

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;
        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;
}

/* 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      */

```

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 chapter “form”.

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.

<code>REQ_CLEAR_PATTERN</code>	- clear pattern buffer
<code>REQ_BACK_PATTERN</code>	- delete last character from pattern buffer
<code>REQ_NEXT_MATCH</code>	- move to next pattern match
<code>REQ_PREV_MATCH</code>	- 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, any time 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 5-5 diagrams this relationship between ETI key values, ETI menu requests, and your application program's menu requests.

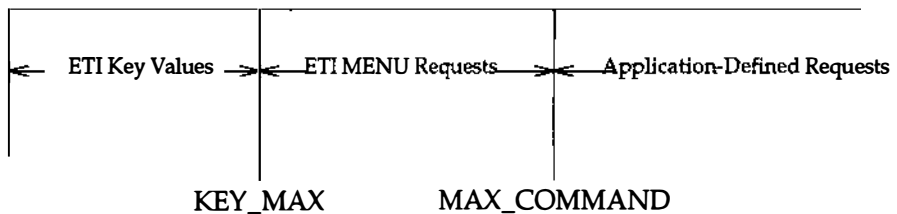


Figure 5-5 Integer ranges for ETI key values and MENU requests

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 5-6 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 5-6 Sample menu output (S)

You have already seen much of the astrological sign program in previous examples. Its function `get_request()`, for instance, appeared in Example 5-8. Example 5-10 shows its remaining routines.

Example 5-10 Sample program calling the menu driver

```

/*
 * menu2: A sample program calling the menu driver
 *
 * To compile: cc menu2.c -lmenu -lcurses
 *
 */

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

#define QUIT (MAX_COMMAND + 1)

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

static void start_curses();      /* initialize curses */
static void end_curses();       /* terminate curses */
static void error ();           /* fatal error handler */
static int get_request ();      /* virtual key mapping */
static void display_menu ();    /* create menu windows and post */
static void erase_menu ();      /* unpost and delete menu windows */

```

```

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",
    (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++);
}

```

```

        /* Initialize curses */
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("\n");
    exit(1);
}

/* define application commands */

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);
}

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;
        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;
}

```

/* 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    */

```

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 **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.

SYNTAX

```

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

Example 5-11 shows how you can use function `set_item_init()` to implement a menu prompting feature as your end user moves from item to item.

Example 5-11 Using an initialization routine to generate item prompts

```

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 */
}

```

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.

SYNTAX

```
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:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- NULL menu pointer or index out of range
<code>E_BAD_STATE</code>	- called from init/term routines
<code>E_NOT_CONNECTED</code>	- 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()`.

SYNTAX

```
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. Example 5-12 illustrates this usage.

Example 5-12 Returning cursor to its correct position for menu driver processing

```
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);
}
```

If function `pos_menu_cursor()` is successful, it returns `E_OK`. In the following error situations, it fails and returns the indicated value:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- null menu pointer
<code>E_NOT_POSTED</code>	- 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()**.

SYNTAX

```
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.

SYNTAX

```
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 Example 5-13 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 Example 5-8.

Example 5-13 Example setting and using a menu user pointer

```

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);
}

```

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 * userptr = "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.

```
SYNTAX

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 <code>REQ_TOGGLE_ITEM</code> request, another is to call <code>set_item_value()</code> . (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.

SYNTAX

```
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

Chapter 6

form

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. Example 6-1 displays a simple form with five fields including two for data entry.

Example 6-1 Sample form display

Sample Form

Field 1: _____
Field 2: _____

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
```

If you want to use the menu or panel routines as well, place the appropriate `-l` option before the option `-lcurses`.

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 Example 6-1.

Some important form terminology

The following terms are helpful in working with ETI form functions:

field	an $m \times n$ 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 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 curses

A sample form application program

Example 6-2 shows the ETI program necessary for producing the form in Example 6-1.

Example 6-2 Code to produce a simple form

```
/*
 * form: A simple form program
 * To compile: cc form.c -lform -lcurses
 */

#include <form.h>
#include <string.h>

FIELD * make_label (frow, fcol, label)
int frow;          /* first row */
int fcol;          /* first column */
char * label;      /* label */
(
    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);
}

```

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(S)` 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
- `fl[6]`, an array of field pointers
- `i`, an index variable, initialized to 0

The first five functions initialize `curses` for 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 ETI, while `exit(0)` terminates the program.

There are many ETI form routines not listed in Example 6-2. 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.

SYNTAX

```
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 $((\text{rows} + \text{nrow}) * \text{cols} + 1)$ 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 Example 6-2 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 later sections, “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:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- null field pointer
<code>E_CONNECTED</code>	- 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.

```
SYNTAX

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:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- null field pointer

Moving a field

ETI provides the following function to move an existing disconnected field to a new location.

```
SYNTAX

int move_field (field, firstrow, firstcol)
FIELD * field;
int firstrow;
int firstcol;
```

Example 6-3 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.

Example 6-3 Example shifting all form fields a given number of rows

```
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;
    }
}
```

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:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- null field or <code>firstrow/firstcol < 0</code>
<code>E_CONNECTED</code>	- 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.

SYNTAX

```
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 making 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 following:

`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 argument pointer in programmer-defined field types, see the section below, "Supporting 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 system 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`. Moreover, you can define your own field types, as described later in the section “Creating and manipulating programmer-defined field types.” The following sections describe all ETI generic field types.

TYPE_ALPHA

The form driver restricts a field of this type to alphabetic data.

SYNTAX

```
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.

SYNTAX

```
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.

SYNTAX

```
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 Example 6-4, which sets the `TYPE_ENUM` of field `color` to a list of colors.

Example 6-4 Setting a field to TYPE_ENUM 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);

```

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.

SYNTAX

```

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(S)** 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.

SYNTAX

```
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(S)` 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_REGEX

This type enables you to determine whether the data entered into a field matches a specific regular expression.

SYNTAX

```
set_field_type (field, TYPE_REGEX, expression);
char * expression; /* regular expression */
```

TYPE_REGEX takes one additional argument, the regular expression. See **regcomp(S)** in the *Programmer's Reference Manual* or the chapter, "lex," in the *Programming Tools 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_REGEX, "^[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_REGEX, "^[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.

SYNTAX

```
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:

<code>NO_JUSTIFICATION</code>	- no justification processing (default)
<code>JUSTIFY_LEFT</code>	- left justify value in field
<code>JUSTIFY_RIGHT</code>	- right justify value in field
<code>JUSTIFY_CENTER</code>	- 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:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- 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 **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.

SYNTAX

```
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(S)** 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.

SYNTAX

```
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.

SYNTAX

```
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 Example 6-5, 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).

Example 6-5 Using the field status to update a database

```

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;
    }
}

```

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.

SYNTAX

```
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.

Example 6-6, 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

Example 6-6 Using the field user pointer to match items

```

#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(S) */

    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);
}

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 */
}

```

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.

SYNTAX

```

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 <code>set_field_buffer()</code> .
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()**.

SYNTAX

```
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 settings 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 function `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.

SYNTAX

```
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.

SYNTAX

```
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

Example 6-7 shows how to create a simple two-page form.

Example 6-7 Creating a 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 */
```

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.

SYNTAX

```
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.

```
SYNTAX

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, (C) the form window is NULL, which by convention means that ETI uses `stdscr` as the form window; and (S) 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.

SYNTAX

```
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.

SYNTAX

```
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 **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 **curses** command output and refresh operations to your form's window, not its subwindow.

Figure 6-1 diagrams the relationship between ETI form functions, your application program, and its form window and subwindow.

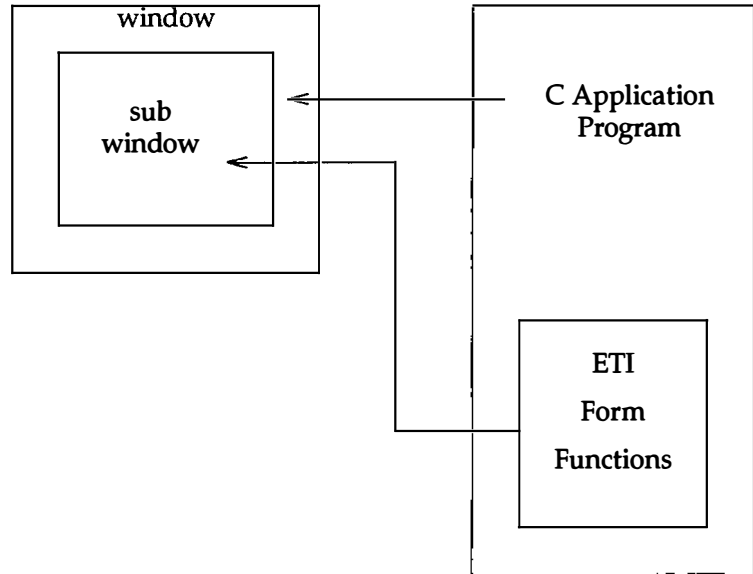


Figure 6-1 Form functions write to subwindow, application to window

Example 6-8 shows how to create a form with a border of the terminal's default vertical and horizontal characters.

Example 6-8 Creating a border around a form

```

/* 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 */
}

```

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:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_POSTED</code>	- 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.

```
SYNTAX

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.

Example 6-9 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 Example 6-8 to create the form’s window and subwindow.

Example 6-9 Posting and unposting a form

```

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 Example 6-8 */

    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
        wrefresh (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);
}

```

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.

```
SYNTAX

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 routine 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, responding to help requests, generating an error message, and so on.

Defining the virtual key mapping

For a sample virtual key mapping, consider Example 6-10 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 section "Calling the form driver."

Example 6-10 A sample key virtualization routine

/* 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  */
```

(Continued on next page.)

Example 6-10 A sample key virtualization routine (Continued)

```

#define QUIT          (MAX_COMMAND + 1)

static int get_request (w)          /* virtual key mapping */
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:    return    REQ_FIRST_FIELD;
        case KEY_LL:    return    REQ_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:    return    REQ_LEFT_CHAR;
        case KEY_RIGHT:    return    REQ_RIGHT_CHAR;
        case KEY_DOWN:    return    REQ_DOWN_CHAR;
        case KEY_UP:    return    REQ_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
                return mode = REQ_INS_MODE;
    }
    return c;
}

```

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

The following requests enable your end-user to navigate or move from page to page on a multi-page form.

<code>REQ_NEXT_PAGE</code>	- move to next page
<code>REQ_PREV_PAGE</code>	- move to previous page
<code>REQ_FIRST_PAGE</code>	- move to first page
<code>REQ_LAST_PAGE</code>	- 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

The following 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

The following requests let your end-user move about inside a field. They may generate implicit scrolling operations on scrollable fields.

<code>REQ_NEXT_CHAR</code>	- move to next character in field
<code>REQ_PREV_CHAR</code>	- move to previous character in field
<code>REQ_NEXT_LINE</code>	- move to next line in field
<code>REQ_PREV_LINE</code>	- move to previous line in field
<code>REQ_NEXT_WORD</code>	- move to next word in field
<code>REQ_PREV_WORD</code>	- move to previous word in field
<code>REQ_BEG_FIELD</code>	- move to beginning of field
<code>REQ_END_FIELD</code>	- move after last character in field
<code>REQ_BEG_LINE</code>	- move to beginning of line
<code>REQ_END_LINE</code>	- move after last character in line
<code>REQ_LEFT_CHAR</code>	- move left in field
<code>REQ_RIGHT_CHAR</code>	- move right in field
<code>REQ_UP_CHAR</code>	- move up in field
<code>REQ_DOWN_CHAR</code>	- 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

The following 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 following requests enable your program to scroll through fields to display this offscreen 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 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 `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, that is, 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 6-2.

Sweepstakes Entry Form		
Last Name	First	Middle
Comments		

Figure 6-2 Sweepstakes form output

You have already seen much of the sweepstakes program in previous examples. Example 6-11 shows its remaining routines.

Example 6-11 An example of form driver usage

```

/*
 * sweeps: An example of form driver usage
 *
 * To compile: cc sweeps.c -lform -lcurses
 *
 * To quit:    type Control-Q
 */

/* This program displays a sweepstakes entry form. */

#include <curses.h>
#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 (); /* create form windows and post */
static void erase_form (); /* unpost and delete form windows */

/* define application commands */

#define QUIT (MAX_COMMAND + 1)

static int get_request (); /* virtual key mapping
static void end_curses();
static void error();
static void dump_form();

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;

FIELD * myf;
char * data;

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;
    }
}

dump_form(form);
addstr("Hit a key\n");
refresh();
getch();

/* terminate form processing */
erase_form (form);
end_curses ();
f = form_fields (form);
free_form (form);
free_fields (f);
exit (0);
}

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",
    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++);
    }

    /* Initialize curses */
    static char * PGM = (char *) 0; /* program name */
    static int CURSES = FALSE;      /* 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("\n");
        exit(1);
    }

    /* "A Sample Key Virtualization Routine" */
    static int get_request (w) /* virtual key mapping */
    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:    return    REQ_FIRST_FIELD;
        case KEY_LL:    return    REQ_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:    return    REQ_LEFT_CHAR;
        case KEY_RIGHT:    return    REQ_RIGHT_CHAR;
        case KEY_DOWN:    return    REQ_DOWN_CHAR;
        case KEY_UP:    return    REQ_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
                return mode = REQ_INS_MODE;
    }
    return c;
}

/* "Posting and Unposting a Form" - an example routine */
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 */

```

```

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
    wrefresh (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);
}

/*
 * dump_form() - This routine was not part of the ETI example. It was
 * added to demonstrate how to get the information back out of the
 * form. This routine would need to be modified to work in a more
 * general case.
 */
static void dump_form(form)
FORM * form;
{
    FIELD ** f;
    int count, i;
    char * data;

    /*
     * field_count - get number of fields connected to given form.
     */
    if((count = field_count(form)) == -1) {
        printf("The form was NULL.\n");
        refresh();
        return;
    }

```

```

    }
    else if (count == 0) {
       printw("The form doesn't have any fields.\n");
        refresh();
        return;
    }
    else {
       printw("The field_count for this form is %d.\n\n", count);
        refresh();
    }
}

/*
 * form_fields()
 * Return a pointer to the array of field pointers defining the
 * form's fields. Returns NULL if no fields are connected to the
 * form or the form pointer is NULL.
 * form_count() above will catch these two conditions and return.
 */
if((f = form_fields(form)) == (FIELD *) 0) {
    printw("No fields connected to the form\n\n");
    refresh();
    return;
}

/*
 * field_buffer() - print out each entry in the field.
 */
for(i=0; i<count; i++) {
    if((data = field_buffer(f[i], 0)) != NULL) {
        printw("f[%d]: '%s'\n", i, data);
        refresh();
    }
}
}

```

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 Example 6-11. (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 **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.

SYNTAX

```
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.

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, that is, after the page changes on a posted form

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, that is, every time the current field changes on a posted form

set_field_term()

The argument **func** to this function is automatically called by the form driver

- just after the field is validated, that is, just before the current field changes on a posted form
- when the form is unposted

set_form_term()

The argument **func** to this function is automatically called by the form driver

- just before every form page operation, that is, 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, Example 6-12 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.

Example 6-12 Sample termination routine that updates a column total

```
#include <form.h>

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 */
}
```

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 Example 6-13. 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.

Example 6-13 Field initialization and termination to highlight current field

```
#include <form.h>

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);
}
```

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.

SYNTAX

```
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."

Example 6-14 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.

Example 6-14 Example manipulating the current 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;
}

```

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.

SYNTAX

```

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."

Example 6-15 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.

Example 6-15 Example changing and checking the form page number

```
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 */
}
```

If function `set_form_page()` encounters an error, it returns one of the following:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- null form or page out of range
<code>E_BAD_STATE</code>	- called from init/term routines
<code>E_INVALID_FIELD</code>	- 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.

SYNTAX

```
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.

Example 6-16 illustrates one use of this function. Function **printpage()** repositions the cursor after it prints the page number in the form window.

Example 6-16 Repositioning the cursor after printing page number

```
#include <form.h>
#include <curses.h>

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);
}
```

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.

SYNTAX

```
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.

Example 6-17 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.

Example 6-17 Pattern match example using form user pointer

```

#include <form.h>

#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 */
}

```

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.

```
SYNTAX

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 processed. 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_fielddtype()` lets you do this.

```
SYNTAX

FIELDTYPE * link_fielddtype(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_fielddtype (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_fielddtype()** 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_fielddtype()** returns your new field type given pointers to these validation functions.

```
SYNTAX

typedef int (* PTF_int) ();

FIELDTYPE * new_fielddtype (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 validation 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. Function `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_fldtype()` is useful for creating field types for specialized applications. For example, Example 6-18 defines a new field type `TYPE_HEX` as a hex number between `0x0000` and `0xffff`.

Example 6-18 Creating a programmer-defined field type

```
#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;
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 */
```

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 hexadecimal 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.

```
SYNTAX

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:

<code>E_SYSTEM_ERROR</code>	- system error
<code>E_BAD_ARGUMENT</code>	- null field type
<code>E_CONNECTED</code>	- 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.

SYNTAX

```
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 <code>set_field_type()</code> and return a pointer to the saved data
copy_arg	duplicate the structure created by <code>make_arg</code>
free_arg	free any storage allocated by <code>make_arg</code> or <code>copy_arg</code>

Function `make_arg` is called automatically when your application program calls `set_field_type()`. It takes one argument, a `va_list *`. (See `varargs(M)` 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`.

Example 6-19 illustrates how you can add padding and range arguments to our TYPE_HEX defined above.

Example 6-19 Creating TYPE_HEX with padding and range arguments

```
/* 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);
}
```

(Continued on next page.)

Example 6-19 Creating TYPE_HEX with padding and range arguments (Continued)

```
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;
    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_fielddtype (fcheck_hex, ccheck_hex);
set_fielddtype_arg (TYPE_HEX, make_hex, copy_hex, free_hex);
```

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.

```
SYNTAX

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.

Example 6-20 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.



Example 6-20 Creating a next choice function for a field type

```

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_fielddtype_choice (TYPE_HEX, next_hex, prev_hex);

```

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_fldtype_choice()** encounters an error, it returns one of the following:

- | | |
|-----------------------------|---|
| <code>E_SYSTEM_ERROR</code> | - system error |
| <code>E_BAD_ARGUMENT</code> | - either field type <code>next_choice</code>
or <code>prev_choice</code> is null |



Chapter 7

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(S)** manual page before you try to use the following ETI features. 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 Release 3.0 or later versions of the **curses(S)** library 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'ed (**|**) 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(S)` 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(S)`.) 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 lowercase 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(S)` 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 `doupdate()`, 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(S)` 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(S)` 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. Example 7-1 shows a typical scenario to output a message to several terminals.

Example 7-1 Sending a message to several terminals

```
for (i=0; i<nterm; i++)
{
    set_term(terms[i]);
    mvaddstr(0, 0, "Important message");
    refresh();
}
```


Chapter 8

terminfo

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 **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 Example 8-1.

Example 8-1 Typical framework of a terminfo program

```
#include <curses.h>
#include <term.h>
...
setupterm( (char*)0, 1, (int*)0 );
...
putp(clear_screen);
...
reset_shell_mode();
exit(0);
```

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(S)**, 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.

When **terminfo** is accessed directly, the **curses** 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(F)**; see **curses(S)** 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 a program that uses ETI" and "Running a program that uses ETI" in the chapter "ETI" 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 **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.

```
/*
 * hilite: A terminfo version of the highlight program.
 * To compile: cc hilite.c -lcurses
 */

#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)
    {
        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 terminal descriptions in the `terminfo` database probably contain strings like `$<20>`, which means to pad for 20 milliseconds (see the section "Specify capabilities" 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**. XX "putchar() function, curses" 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** routines instead of **terminfo** 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. Details such as this are precisely why the **curses** is recommended over direct access to **terminfo**. **curses** takes care of terminals with different methods of underlining and other terminal functions. Programs that use **terminfo** directly 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(S)**) 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(C)**, 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 (|). 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(M)** 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(F)** 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 description. Comments can be included in the description by putting a **#** at the beginning of the line.

The **terminfo(F)** manual page has a complete list of the capabilities you can use in a terminal description. This list contains the name of the capability, 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 descriptions 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:

- ^ 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 `^G`.
- `\E` or `\e` These characters followed by another character show an escape instruction. An entry of `\EC` would transmit to the terminal as `<Esc>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(F)** 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 **<Esc>D (sms0)**.
- Exiting reverse video mode is an **<Esc>Z (rms0)**.
- A clear to the end of a line sequence is an **<Esc>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 `<Esc>[ A` (**kcuu1**).
- The down arrow key generates an `<Esc>[ B` (**kcud1**).
- The right arrow key generates an `<Esc>[ C` (**kcuf1**).
- The left arrow key generates an `<Esc>[ D` (**kcub1**).
- The home key generates an `<Esc>[ 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(S)** 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(F)** 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 `<Esc>[` 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 `'<Esc>[ 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
<code>\E[</code>	output <code><Esc>[</code>
<code>%i</code>	increment the two arguments
<code>%p1</code>	push the 1st argument (the row) onto the stack
<code>%d</code>	output the row as a decimal
<code>;</code>	output a semi-colon
<code>%p2</code>	push the 2nd argument (the column) onto the stack
<code>%d</code>	output the column as a decimal
<code>H</code>	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(F)` 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(C)** 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(ADM)** manual page 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(C)**) 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(C)** 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 [-Ttype] capname
```

The type of terminal you are requesting information about is identified with the **-Ttype** 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(C)** manual page 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(ADM)** 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

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**(ADM) (see the *User's Reference*) command converts **termcap**(F) descriptions to **terminfo**(F) 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 **-lterm lib** option in the **cc** command line, those programs will still be functional. However, these options should be replaced with the **-lcurses** option.

Chapter 9

Program examples

The following programs demonstrate uses of **curses** 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 entitled “curses”.

Second, it also uses some **curses** routines that we have not discussed. For example, the function to write out a file uses the **mvinch(S)** 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(S)** 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.
 *
 * To compile: cc editor.c -lcurses
 */

#include <stdio.h>
#include <curses.h>

#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;

/* 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;

        /* 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;

    stdout();
    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(S)`). 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 filter that turns
 * \U...\N, and \B...\N sequences into
 * highlighted output, allowing words to be
 * displayed underlined or in bold respectively.
 *
 * To compile: cc hilight.c -lcurses
 */

#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)
    {
        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 an input file 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.

```

/*
 * scatter: A program that scatters input on the terminal.
 *
 * To compile: cc scatter.c -lcurses -o scatter
 * To use: scatter < scatter.c
 */

#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(S)` routine, which we also have not discussed, is called to allow insert and delete line. (See `curses(S)` for more information about these routines). Also notice that `clrtoeol(S)` and `clrtoobot()` 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.

```

/*
 * show: A program that displays a screenful at a time.
 *
 * To compile: cc show.c -lpanel -lcurses
 */

#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))
            {
                clrtoebot();
                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 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.

```

/*
 * two: A program that displays on two terminals
 *
 * To compile: cc two.c -lpanel -lcurses
 */

#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);
    }
    fd = fopen(argv[3], "r");
    fdyou = fopen(argv[1], "w+");
    signal(SIGINT, done); /* die gracefully */

    /* initialize both ttys */
    me = newterm(getenv("TERM"), stdout, stdin);
    you=newterm(argv[2],fdyou,fdyou);

    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) {
            clrtoebot();
            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(S)` for that window causes it to be written over the `stdscr` image on the terminal screen. Calling `refresh(S)` 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(S)`) 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(S)` for the new window to get it completely written out.

```

/*
 * window: A sample of one window overlaying another
 *
 * To compile: cc window.c -lcurses
 */

#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);

    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.

```

/*
 * colors: A simple curses program
 * Displays colored text in two windows.
 * Red text on blue background.
 * Yellow text on magenta background.
 * To compile: cc colors.c -lcurses
 */

#include <curses.h>

#define PAIR1 1
#define PAIR2 2

main()
{
    WINDOW *win1, *win2;

    initscr();
    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();
}

```



Index

A

addch() function, curses, 22, 31
 Adding
 character, curses, 27
 character to standard screen, curses, 22
 line, curses, 28
 string to standard screen, curses, 23
 addstr() function, curses, 23
 Application-defined commands, ETI, 111, 180
 Array, creating, ETI, 80
 attroff() function, curses, 32
 attron() function, curses, 32
 attr() function, curses, 32
 attrset() function, curses, 31-32, 243
 autoflush(bf) function, curses, 51

B

beep() function, curses, 30
 Bold highlighting, curses, 243
 Border
 form, ETI, 169
 panel, ETI, 64
 bottom_panel() function, ETI, 64, 66, 68
 Box drawing, curses, 54
 box() function, curses, 54, 216
 Buffers, ETI, 139

C

Calling
 form driver, ETI, 180
 menu driver, ETI, 111
 cbreak function, curses, 42
 Changing
 default values for menu attributes, ETI, 91
 form page, ETI, 195
 item attribute default value, ETI, 86
 menu display attributes, ETI, 102
 menu mark string, ETI, 97
 panel windows, 65
 Changing and fetching, pattern buffer, ETI, 126

Character

 attributes, curses, 31-32
 deleting, curses, 28
 inserting, curses, 27
 Checking item visibility, ETI, 85
 Choice requests, ETI, 180
 chtype() function, curses, 28, 31
 clear() function, curses, 29
 Clearing screen, curses, 29-30
 clearok() function, curses, 55
 clear-screen flag set/clear, curses, 55
 clear_screen() function, curses, 222
 clrtobot() function, curses, 30
 clrtoeol() function, curses, 30
 Color
 changing definitions, curses, 36
 definition change, curses, 39
 manipulation, curses, 33
 pair definition change, curses, 38
 pairs table, curses, 35
 restoring original, curses, 38
 Colors program example, curses, 252
 Commands, application defined, ETI, 180
 compare() function, ETI, 198
 Compile options for ETI, 17
 Compiler, tic, curses, 232
 Compiling, database entries, curses, 232
 Compiling and linking
 curses programs, 21
 form programs, ETI, 133
 menu programs, ETI, 75
 panel programs, ETI, 63
 Compiling and running, terminfo program, 222
 Connecting items to a menu, ETI, 76
 Copying one window to another, curses, 51-52
 Counting menu items, ETI, 91
 Creating
 form, example, ETI, 163
 pad, curses, 61
 panels, ETI, 63
 subpad, curses, 61
 windows, curses, 48

Creating and freeing

- menu items, 80
- menus, ETI, 88

Current

- position, curses, 54
- screen. *See* `curscr`
- terminal, curses, 218

`current_field()` function, ETI, 194

`current_item()` function, ETI, 122

`curscr` default window, ETI, 15

curses

- character attributes, 31
- color and video attributes, ETI, 102
- color manipulation, 33
- compiling and linking programs, 21
- controlling the terminal, 40
- `curses.h` header file, 14, 21, 222
- drawing lines and other graphics, 215
- introduction, 21
- library, ETI, 9, 14
- pads, 47
- routines for panels, ETI, 64
- routines, return values, 21
- screen manipulation routines, 21
- subpads, 47
- subwindows, 47
- terminating routine, ETI, 79
- using soft labels, 216
- using the standard screen, 22
- windows, creating and using, 48
- windows introduced, 47
- working with more than one terminal, 218

Cursor

- flag set/clear, curses, 55
- movement and action combined, curses, 27
- movement, curses, 43
- movement optimization, ETI, 10
- position change, curses, 26
- positioning, ETI, 196
- positioning menu cursor, ETI, 125

D

Database

- `termcap`, 235
- `terminfo`, 235

Debugging program that uses ETI, 20

Default values

- field attributes, ETI, 143

Default values (*continued*)

- menu attributes, ETI, 91

Default window

- `curscr`, ETI, 15
- curses, 22
- `stdscr`, ETI, 15

Defining the virtual key mapping, ETI, 173

`delch()` function, curses, 28, 237

`deleteln()` function, curses, 29

Deleting

- character, curses, 28
- line, curses, 29
- panels, ETI, 74

`del_panel()` function, ETI, 74

`delwin()` function, ETI, 74

Dimensions

- forms, ETI, 166
- menu, ETI, 98
- menus, ETI, 92

`display_deck` function, ETI, 74

`display_form()` function, ETI, 170

Displaying

- attributes, ETI, 102
- forms, ETI, 166
- menus, ETI, 92

`doupdate()` function, curses, 56

`doupdate()` function, ETI, 64, 67

Drawing

- box, curses, 54, 216
- lines and other graphics, curses, 215

Driver

- calling menu driver, ETI, 111
- form, ETI, 172

`dup_field()` function, ETI, 140

E

`echo()` function, curses, 42

Echoing characters to the screen, 42

Editor program, curses, 237

`endwin()` function

- ETI, 17, 79

`enter_bold_mode()` function, curses, 225

`enter_underline_mode()` function, curses, 225

`erase()` function, curses, 29

Erase menu, ETI, 79

`erase_form()` function, ETI, 170

ERR message, curses, 21

ETI

- compile options, 17
- cursor movement optimization, 10
- form requests, 175
- input option settings, 40
- libraries, 9, 14
- menu requests, 108
- program debugging, 20
- program execution, 19
- program requirements, 12
- screen update optimization, 10
- terminfo and termcap, 10
- terminology, 76
- Executing program with ETI, 19
- exit() function, ETI, 79
- exit_attribute_mode() function, curses, 225

F

Fetching and changing

- current item, ETI, 122
- menu display attributes, ETI, 102
- menu items, ETI, 90
- top row, ETI, 123

Fetching item names and descriptions, ETI, 83

Field

- counting, ETI, 165
- editing requests, ETI, 178
- initialization, ETI, 190
- location change, ETI, 142
- navigation requests, ETI, 176-177
- option manipulation, ETI, 158
- setting and reading buffers, ETI, 152
- setting and reading status, ETI, 154
- setting and reading user pointer, ETI, 156
- size and location, ETI, 141
- status used to update database, ETI, 155
- termination, ETI, 190
- type with validation functions, ETI, 203
- validation requests, ETI, 180
- field_arg() function, ETI, 144
- field_buffer() function, ETI, 153, 155
- field_count() function, ETI, 165
- field_index() function, ETI, 194
- field_info() function, ETI, 141
- field_status() function, ETI, 154-155
- field_type() function, ETI, 144
- File, header, ETI, 14
- find_field() function, ETI, 158

find_match() function, ETI, 156

first_field() function, ETI, 194

Flags, saving and restoring, curses, 43

flash() function, curses, 30, 237

form

- changing default attributes, 166
- compiling and linking form program, 133
- creating and freeing fields, 139
- creating and freeing forms, 161
- creating and manipulating programmer-defined field types, 202
- creating borders, 169
- displaying, 166
- driver call, ETI, 180
- driver processing, 172
- field buffers, 152
- form.h header file, 14
- form.h header file, ETI, 133, 173
- initialization, ETI, 190
- introduced, ETI, 133
- library, ETI, 9, 14
- manipulating field attributes, 141
- manipulating field options, 158
- manipulating form attributes, 164
- setting and fetching options, 200
- setting field foreground, background and pad character, 151
- termination, ETI, 190
- terminology, 134
- writing form programs, 134

form field

- TYPE_ALNUM, 145
- TYPE_ALPHA, 145
- TYPE_ENUM, 146
- TYPE_INTEGER, 147
- TYPE_NUMERIC, 148
- TYPE_REGEX, 149

Format, menu, ETI, 93

form_driver() function, ETI, 172

form_page() function, ETI, 195

formsfield status flags, 152

formsfield user pointers, 152

form_userptr(), ETI, 198

free_field_id() function, ETI, 156

free_form() function, ETI, 138, 163

Freeing

- item, ETI, 76
- menu, ETI, 76
- free_item() function, ETI, 79, 81, 89

free_menu() function, ETI, 79, 89
Function key programming, curses, 221
Functions, ETI, 9

G

getch function, curses, 24
getdim(win,y,x) function, curses, 51
getorg(win,y,x) function, curses, 51
get_request() function, ETI, 107, 173
getstr() function, curses, 25
getxy() function, curses, 54
Graphics, curses, 215

H

Header file
ETI, 14
curses.h, 14, 21, 222
form.h, 14
menu.h, 14
panel.h, 14
term.h, 222
Hidden panel
check, ETI, 69
reinstatement, ETI, 70
hide_panel() function, ETI, 67-69, 74
Highlight
attribute, curses, 33
program example, curses, 243

I

init_color() function, curses, 39
Initialization
generate item prompts, ETI, 120
item and menu, ETI, 118
routines, curses, 21
init_pair() function, curses, 38
initscr() function, ETI, 14
initscr() function, ETI, 137
Input
from pads, curses, 51
from windows, curses, 50
keyboard, curses, 22
option settings, ETI, 40
routines, curses, 21, 51
insch() function, curses, 27, 237

Inserting
character, curses, 27
line, curses, 28
insertln() function, curses, 28, 237
iscuroff() function, curses, 51
is_standout(win) function, curses, 51
Item
changing attribute default value, ETI, 86
checking visibility, ETI, 85
counting menu items, 91
descriptions, ETI, 83
fetching and changing, ETI, 90
fetching and changing current item, 122
fetching names and descriptions, 83
initialization routines, 118
introduced, ETI, 76
names, ETI, 83
navigation request, ETI, 108
options, ETI, 83
setting options, 84
setting user pointer, ETI, 86
value manipulation, ETI, 82
visibility, ETI, 83
item_count() function, ETI, 91
item_description function, ETI, 83
item_index() function, ETI, 122
item_name() function, ETI, 83
item_opts() function, ETI, 84
item_userptr() function, ETI, 86
item_value() function, ETI, 82
item_visible() function, ETI, 85

J

Justifying data in a field, ETI, 149

K

Key virtualization defining, ETI, 107
Keyboard
capabilities, curses, 231
reading a character, curses, 24
reading a string, curses, 25

L

Labels, curses, 216

leaveok() function, curses, 55

Library

ETI introduced, 9

TAM transition, ETI, 12, 19

corresponding header files, ETI, 14

curses, ETI, 9, 14

form, ETI, 9, 14

menu, ETI, 9, 14

panel, ETI, 9, 14

Line

deleting, curses, 29

inserting, curses, 28

setting newline mode, curses, 42

link_field() function, ETI, 140

link_fieldtype() function, ETI, 202

longname() function, curses, 44

M

Macros, ETI, 9

main() function, ETI, 198

make_field() function, ETI, 137

make_label() function, ETI, 137

Manipulating

current field, ETI, 194

form attributes, ETI, 164

item value, ETI, 82

menu attributes, ETI, 90

menu user pointer, ETI, 127

Mark string, ETI, 97

Matching items with field user pointer,
ETI, 157

menu

ETI, 76

creating and freeing, 88

creating and freeing menu items, 80

displaying, ETI, 92

driver processing, 106

fetching and changing menu items, 90

introduced, 75

library, ETI, 9, 14

manipulating item attributes, 83

manipulating menu attributes, 90

manipulating user pointer, 127

menu.h header file, 14

menu.h header file, ETI, 79

multi-valued, 82

setting and fetching options, 129

single-valued, 82

menu (continued)

subwindow, ETI, 76, 99

window, ETI, 76, 99

writing menu programs in ETI, 76

menu_driver() function, ETI, 106, 172

menu_opts() function, ETI, 129

menu_opts_off() function, ETI, 82

Message, pop-up, ETI, 74

move() function, curses, 26-27, 237

move_panel() function, ETI, 66

Moving

field, ETI, 142

form page to form page, ETI, 175

from field to field, ETI, 176

panel windows on screen, ETI, 66

window, curses, 53

within a field, ETI, 177

Multiple

selection menu, ETI, 82

terminal use, curses, 218

window program example, curses, 251

Multi-valued menu selection, ETI, 109

mvaddstr() function, curses, 237

mvcur function, curses, 43

mvinch() function, curses, 237

mvwin() function, curses, 53

mvwinshc() function, ETI, 64

N

new_field() function, ETI, 137, 152, 166

new_fieldtype() function, ETI, 203

new_form() function, ETI, 138, 161, 163, 166

new_item() function, ETI, 79-80, 86

Newline mode, curses, 42

new_menu() function, ETI, 79, 88

newpad() function, curses, 61

new_panel() function, ETI, 63

newwin() function, curses, 48

Next and previous choice functions, ETI, 210

nl() function, curses, 42

nocrbreak function, curses, 42

noecho() function

ETI, 137

curses, 42

nonl() function

ETI, 137

curses, 42

noraw() function, curses, 43

NULL pointer, ETI, 76

O

Offscreen rows, ETI, 139
 OK message, curses, 21
 O_ONEVALUE menu option, ETI, 82
 Optimization
 cursor movement, ETI, 10
 screen update, ETI, 10
 screen updates, ETI, 15
 Option
 menu, ETI, 82
 menu, setting and fetching, ETI, 129
 setting and fetching form options, ETI, 200
 O_ROWMAJOR option, ETI, 92
 O_SHOWDESC option, ETI, 92
 Output
 routines, curses, 21
 to pads, curses, 51
 to standard screen, curses, 22
 to windows, curses, 50
 Overlapping panels, ETI, 64
 overlay() function, curses, 51
 Overlaying windows, curses, 51
 overwrite() function, curses, 52
 Overwriting a screen, curses, 52

P

pad
 creating, curses, 61
 curses, 47
 input output routines, curses, 51
 refreshing, curses, 62
 Page
 changing and checking number, ETI, 196
 navigation requests, ETI, 175
 panel
 compiling and linking programs, ETI, 63
 creating, 63
 deck, ETI, 64
 deleting, 74
 elementary panel window operations, 64
 fetching panels from above or below, 70
 introduced, ETI, 63
 library, ETI, 9, 14
 making panels invisible, ETI, 68
 moving on screen, ETI, 66
 moving to top/bottom of deck, ETI, 66
 setting and fetching panel user pointer, 71
 updating on the screen, 67
 panel_above() function, ETI, 70

panel_below() function, ETI, 70
 panel.h header file, 14
 panel_hidden() function, ETI, 69
 panel_userptr() function, ETI, 71
 panel_window() function, ETI, 64
 Parameter string capabilities, curses, 231
 Pattern
 buffer, changing and fetching, ETI, 126
 buffer, requests, ETI, 110
 match with form user pointer, ETI, 198
 matching, ETI, 76, 80
 Pointer
 manipulating menu user pointer, ETI, 127
 to panel windows, ETI, 64
 Pop-up message, ETI, 74
 pos_form_cursor() function, ETI, 197
 Position, current, curses, 54
 Positioning
 form cursor, 196
 menu cursor, ETI, 125
 pos_menu_cursor() function, ETI, 125
 post_form() function, ETI, 138, 171
 Posting, menu, ETI, 76
 Posting/unposting, forms, ETI, 170
 Posting/unposting menus, ETI, 104
 post_menu() function, ETI, 104
 prefresh() function, curses, 62
 Previous and next choice functions, ETI, 210
 Primitives, curses, 221
 Printing
 characters to standard screen, curses, 23
 numbers to standard screen, curses, 23
 strings to standard screen, curses, 23
 printf() function, curses, 23
 Processing
 menu driver, ETI, 106
 menu items, ETI, 83
 process_menu() function, ETI, 83
 Program requirements, ETI, 12
 Programmer-defined field type
 argument support, 207
 creating and manipulating, ETI, 202
 freeing, ETI, 206
 supporting, 207
 Programming a function key, curses, 221
 Programs, writing form programs, 134
 putp() function, curses, 222, 225

R

raw() function
 ETI, 137
 curses, 43
 Reading
 character from keyboard, curses, 24-25
 field buffers, ETI, 152
 field status, ETI, 154
 string from keyboard, curses, 25
 to a window, curses, 50
 refresh() function
 curses, 22, 51, 53, 55-56
 refresh() function, ETI, 14
 Refreshing
 pad, curses, 62
 screen, ETI, 79
 windows, curses, 48
 replace_panel() function, ETI, 65
 Repositioning cursor, ETI, 197
 reset_shell_mode() function, curses, 222
 resetty function, curses, 43
 Reshape panel windows, 65
 Restoring terminal settings, ETI, 17
 Return values, curses routines, 21
 Rows, ETI, 139
 Running program that uses ETI, 19

S

savetty function, curses, 43
 scale_form() function, ETI, 166-167
 scale_menu() function, ETI, 98
 Scaling form size, ETI, 167
 scanw() function, curses, 25
 Scatter program example, curses, 244
 Screen
 capabilities, curses, 230
 changing color definitions, curses, 36
 character attributes, curses, 31-32
 clearing, curses, 29
 color, curses, 33, 35
 creating a pad, curses, 61
 creating a subpad, curses, 61
 current position, curses, 54
 cursor movement, 43
 cursor movement and action combined, curses, 27
 deleting a character, curses, 28
 deleting a line, curses, 29
 drawing a box, curses, 54

Screen (continued)

 inserting a character, curses, 27
 inserting a line, curses, 28
 management, curses, 221
 manipulation routines, curses, 21
 moving a window, curses, 53
 moving cursor position, curses, 26
 overwriting, curses, 52
 panels. *See* panel, ETI
 refreshing a pad, curses, 62
 relationship to window, curses, 57
 scrolling a window, curses, 56
 standard, curses, 22
 standout mode, curses, 33
 touching a window, curses, 53
 update optimization, ETI, 10
 updating, curses, 56
 updating with refresh() and wrefresh(), 15
 using COLOR_PAIR(n), curses, 36
 writing editor, curses, 237
 scroll flag set/clear, curses, 55
 scroll() function, curses, 56
 Scrolling
 menu, ETI, 109
 requests, ETI, 179
 window, curses, 56
 scrollok() function, curses, 55, 243
 Sending signals from the terminal, curses, 30
 set_current_field() function, ETI, 191, 194
 set_current_item() function, ETI, 122, 124
 set_field() function, ETI, 144
 set_field_back() function, ETI, 151
 set_field_buffer() function, ETI, 138, 144, 152
 set_field_fore() function, ETI, 151
 set_field_id() function, ETI, 156
 set_field_init() function, ETI, 191
 set_field_just() function, ETI, 149
 set_field_opts() function, ETI, 138, 158
 set_field_pad() function, ETI, 151
 set_field_status() function, ETI, 155
 set_field_term() function, ETI, 191
 set_field_type() function, ETI, 144, 202
 set_fieldtype_arg() function, ETI, 207
 set_field_usrptr() function, ETI, 158
 set_form_fields() function, ETI, 164
 set_form_init() function, ETI, 191
 set_form_page() function, ETI, 191, 195
 set_form_sub() function, ETI, 170
 set_form_term() function, ETI, 191
 set_form_userptr() function, ETI, 198

set_form_win() function, ETI, 170
set_item_init() function, ETI, 120
set_item_opts() function, ETI, 84
set_item_term() function, ETI, 120
set_item_userptr() function, ETI, 86
set_item_value() function, ETI, 82
set_menu_pattern() function, ETI, 126
set_menu_format() function, ETI, 93
set_menu_init() function, ETI, 120
set_menu_items() function, ETI, 90
set_menu_mark() function, ETI, 97
set_menu_opts() function, ETI, 82, 129
set_menu_pattern function, ETI, 124
set_menu_sub() function, ETI, 92, 99
set_menu_term() function, ETI, 120
set_menu_usrptr() function, ETI, 127
set_menu_win() function, ETI, 92, 99
set_new_page() function, ETI, 162
set_panel_userptr() function, ETI, 71, 74
setterm() function, curses, 44
Setting
 character attributes, curses, 32
 field type for validation, ETI, 144
 item options, ETI, 84
 item user pointer, ETI, 86
 window flags, curses, 55
Setting and fetching
 form options, ETI, 200
 form user pointer, 198
 menu options, ETI, 129
Setting and reading
 field buffers, ETI, 152
 field status, ETI, 154
 field user pointer, ETI, 156
set_top_item() function, ETI, 124
set_top_row() function, ETI, 123
setupterm() function, curses, 222
Show program example, curses, 246
show_panel() function, ETI, 64, 68, 70, 74
Signals sent from terminal, curses, 30
Single selection menu, ETI, 82
Size of forms, ETI, 166
sleep() function, ETI, 79, 138
Specifying menu format, ETI, 93
Standard screen
 adding a character, curses, 22
 adding a string, curses, 23
 curses, 22, 48
 printing strings, characters, numbers,
 curses, 23
standend() function, curses, 33

standout() function, curses, 33
start_color() function, curses, 38
stdscr
 character attributes, curses, 31
 default window, ETI, 15
 default window, curses, 22
 panels, ETI, 63
 terminal screen relationship, 16
String, special purpose, curses, 221
Structure, curses, 51
subpad
 creating, curses, 61
 curses, 47
subpad() function, curses, 61
subwin() function, curses, 49
Subwindows
 associated with forms, ETI, 168
 associated with menus, ETI, 99
 creation, curses, 49
 curses, 47
Syntax, compile options for ETI, 17

T

TAM

 screen management package, 19
 transition library, ETI, 12, 19
tam.h header file, ETI, 19
Temporary window, curses, 52-53
TERM environment variable, ETI, 14
termcap database, ETI, 10
Terminal
 capabilities database, ETI, 10
 control, ETI input options, 40
 control, curses, 40
 cursor movement, 43
 description database, ETI, 10
 description test, curses, 233
 descriptions, curses, 225
 ensuring correct settings, 45
 manipulation, curses, 221
 name, curses, 226
 reading name, curses, 44
 saving and restoring flags, curses, 43
 screen and stdscr relationship, 16
 setting restore, ETI, 17
 signals, curses, 30
 type, ETI, 14
 type setting, curses, 44
 using more than one, curses, 218

Terminal Access Method. *See* TAM

Terminal capabilities

basic, curses, 230

curses, 227

description, curses, 228

keyboard-entered, 231

parameter string, 231

screen-oriented, 230

Termination

item and menu, ETI, 118

routines, curses, 21

terminfo

comparing terminfo descriptions, 234

converting termcap description to terminfo
description, 235

database, 235

database, ETI, 10

introduced, curses, 221

printing terminfo descriptions, 234

program requirements, 222

working with terminfo database, 225

term.n header file, 222

Testing terminal description, curses, 233

Text standout on screen, curses, 221

tic compiler, curses, 232

Title for panel, ETI, 64

top_panel() function, ETI, 64, 66, 68

top_row() function, ETI, 123

Touching a window, curses, 53

touchwin() function, curses, 53

touchwin() function, ETI, 67

tparm() function, curses, 231

tputs() function, curses, 222, 224

tscroll(win, bf) function, curses, 51

Two terminal program example, curses, 248

U

underline_char() function, curses, 225

Underlining, curses, 243

unpost_form() function, ETI, 138, 171

Unposting menu, ETI, 76, 79

Unsetting character attributes, curses, 32

update_panels() function, ETI, 67, 69, 74

Updating

contents of a window, curses, 56

panels on the screen, 67

upost_menu() function, ETI, 104

V

vidattr() function, curses, 225

Virtual key mapping, ETI, 173

Visibility

menu item, ETI, 85

panels, ETI, 64

W

waddch() function, curses, 215

wgetch() function, ETI, 68

wgrech() function, ETI, 137

while() function, ETI, 138

Window

associated with forms, ETI, 168

associated with menus, ETI, 99

copying, curses, 51

default, ETI, 15

default, curses, 22

drawing a box, curses, 54

input output routines, 50

more efficient refresh, curses, 56

moving, curses, 53

overlying, curses, 51

panels. *See* panel, ETI

program example, curses, 251

refresh() function, ETI, 14

relationship to terminal screen, curses, 57

routines, curses, 48

scrolling, curses, 56

setting flags, curses, 55

size, ETI, 92

structure, curses, 51

temporary, curses, 52

touching, curses, 53

update contents, curses, 56

writing from and reading to, curses, 50

wnoutrefresh() function, curses, 56, 237

wnoutrefresh() function, ETI, 67

Working with terminfo database, 225

wrefresh() function, curses, 48, 51, 56

wrefresh() function, ETI, 14-15, 68

Writing

filter, curses, 221

from a window, curses, 50

menu programs in ETI, 76

screen editor, curses, 237

Please help us to write computer manuals that meet your needs by completing this form. Please post the completed form to the Technical Publications Research Coordinator nearest you: The Santa Cruz Operation, Ltd., Croxley Centre, Hatters Lane, Watford WD1 8YN, United Kingdom; The Santa Cruz Operation, Inc., 400 Encinal Street, P.O. Box 1900, Santa Cruz, California 95061, USA or SCO Canada, Inc., 130 Bloor Street West, 10th Floor, Toronto, Ontario, Canada M5S 1N5.

Volume title: _____
(Copy this from the title page of the manual, for example, SCO UNIX Operating System User's Guide)

Product: _____
(for example, SCO UNIX System V Release 3.2 Operating System Version 4.0)

How long have you used this product?

- ☐ Less than one month ☐ Less than six months ☐ Less than one year
☐ 1 to 2 years ☐ More than 2 years

How much have you read of this manual?

- ☐ Entire manual ☐ Specific chapters ☐ Used only for reference

	Agree			Disagree		
The software was fully and accurately described	☐	☐	☐	☐	☐	☐
The manual was well organized	☐	☐	☐	☐	☐	☐
The writing was at an appropriate technical level (neither too complicated nor too simple)	☐	☐	☐	☐	☐	☐
It was easy to find the information I was looking for	☐	☐	☐	☐	☐	☐
Examples were clear and easy to follow	☐	☐	☐	☐	☐	☐
Illustrations added to my understanding of the software	☐	☐	☐	☐	☐	☐
I liked the page design of the manual	☐	☐	☐	☐	☐	☐

If you have specific comments or if you have found specific inaccuracies, please report these on the back of this form or on a separate sheet of paper. In the case of inaccuracies, please list the relevant page number.

May we contact you further about how to improve SCO UNIX documentation?
If so, please supply the following details:

Name _____ Position _____

Company _____

Address _____

City & Post/Zip Code _____

Country _____

Telephone _____ Facsimile _____

10 december 1991



BH01 406P000

56756

SCO UNIX Development System

User Interfaces Guide



AU01410P000