# Character User Interface Programming

CONCURRENT COMPUTER CORPORATION™

0890424-000

February 1997

The operating system name has been changed to PowerMAX OS™

| Revision History: | Level: | Effective With: |
|---|---|---|
| Original Release -- July 1994 | 000 | *Secure*/Power UNIX r1.1 |

# Preface

## Introduction

The *Character User Interface Programming* is for application developers who want to develop a menu- and form-based interface that operates on ASCII character terminals running on UNIX® System V Release 4.2 and later. Existing applications can be adapted to a character user interface front-end, and new applications can be designed from the start to take advantage of the screen management capabilities of FMLI and ETI.

*FMLI* is a high-level programmer interface for creating menus, forms, and text frames that enforce a well-defined look and feel policy. An application developer, having defined frames (menus, forms, and text) in files, is free from having to program their display and user interactions. The shell-like language is processed by an interpreter and allows the developer to specify menu and form placement. FMLI allows application developers to customize specific applications easily and quickly without writing in C language code.

*ETI* is a set of screen management library subroutines (built on **`curses`**) that promote fast development of application programs that manipulate windows, panels, menus, and forms. ETI also includes functions to define help, error and other types of messages, and to display, and change messages quickly and easily. It is a C language toolkit used to build user interfaces for applications. ETI allows the developer to design a unique/customized user interface.

This guide tells you how to use the Form and Menu Language Interpreter (FMLI) and the Extended Terminal Interface (ETI) software development tools to write such user interfaces for your applications. It assumes the reader has a working knowledge of UNIX System V, shell programming, and/or C Language programming.

**NOTE**

This guide is not intended to be an introduction to UNIX System V, UNIX System shell programming, or C Language programming. For an introduction to shell programming and C Language programming, see “Referenced Publications” on page -vi.

## Who This Guide Is For

This guide is written for programmers developing UNIX System V applications with interfaces that operate on display devices using only standard characters. A working knowledge of the UNIX system and shell programming is assumed. (See the UNIX System V *User's Guide* for detailed information on these topics.)

## Scope of Manual

Chapter 1 through Chapter 4 describe the Form and Menu Language, and tell how to use it to write descriptions of the forms, menus, and text frames that make up your user interface. These chapters explain how you can make best use of the screen management capabilities that FMLI provides for you, and how you can customize the default appearance and functionality of a user interface written with FMLI.

Chapter 5 through Chapter 13 describe the Extended Terminal Interface, and tell how to use it to write screen management programs on a UNIX system. These chapters explain how to use the high-level library routines to build panels, menus and forms. They also describe how these routines relate to low-level **`curses`** routines and the **`terminfo`** database.

## Syntax Notation

The following typographical conventions are used in this guide:

- The logical values "true" and "false" are represented in the text by the words TRUE and FALSE, shown in all capital letters. Boolean descriptors must evaluate to either TRUE or FALSE, where
  - FALSE means the literal word "false," irrespective of case, or a non-zero return code.
  - TRUE means any value other than those defined for FALSE.
- Literal elements of computer input and output, including user input, program code, UNIX system command names, FMLI command and built-in utility names, and other elements of the Form and Menu Language are shown in `constant-width typeface`.
- Substitutable elements of command lines and of elements of the Form and Menu Language are shown in *italic typeface*.
- Comments in a screen display—that is, text that is not computer output but is an aside from the author to the reader—are shown in *italic typeface*, as in the following example:

```
      .
      .
      .
      command interaction
      .
      .
      .
Press ENTER to continue
```

- Named keys are shown in a representation of a hard key. This includes keys such as ESCAPE or DEL, and the function keys F1 through F8.
- Alternative keystroke sequences are also shown in a hard key representation. For example, the alternative keystroke sequence for the named key

DEL is CTRL-x. That means the user must hold down the CTRL key while pressing x.

Longer alternative keystroke sequences are shown as a sequence of hard key representations. For example, the alternative keystroke sequence for the named key F3 is CTRL-f 3. That means the user must hold down CTRL while pressing f, then press 3.

- Screen labels for function keys are also shown as hard key representations. For example, when a menu is the active frame on the screen, function key F1 has the screen label HELP.
- Depending on the keyboard being used, the carriage-return key may be called ENTER, RETURN, or something else. Throughout this guide the ENTER key is used to represent the carriage-return key. However, if a keyboard only has a RETURN key, use it or CTRL-m instead.
- When command syntax is described, the following notation conventions are used (especially in the FMLI manual pages in section 1F):
  - Literal elements of a command (including command names themselves) are shown in `constant-width typeface`.
  - Substitutable arguments to commands are shown in *italic typeface*.
  - Square brackets ([ ]) around an argument indicate that the argument is optional.
  - Ellipses (`. . .`) are used to show that the previous argument may be repeated.
- In program text, the major ETI data types appear in uppercase. They are

| | |
|---|---|
| `WINDOW` | A rectangular area of the screen treated as a unit |
| `PANEL` | A window with relations of depth to other windows so that regions hidden behind other windows are invisible |
| `ITEM` | A character string consisting of a name and an optional description |
| `MENU` | A screen display that presents a set of items from which the user chooses one or more, depending on the type of menu |
| `FIELD` | An *m* x *n* block of form character positions 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 SYNOPSIS. The first line of the SYNOPSIS proper describes the routine, while the following lines describe its arguments. On each line, the type of the return value or arguments precedes their names. As an example, consider

**SYNOPSIS**

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

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

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

## Referenced Publications

The following publications are referenced in this document:

| | |
|---|---|
| 0890428 | User's Guide |
| 0891019 | Concurrent C Reference Manual |

# Contents

## Screens

## Tables

# 1
# Introduction to FMLI

# 1 Introduction to FMLI

## Introduction

This chapter describes the Form and Menu Language Interpreter at a general level. There are five major sections:

- “What Is FMLI?” on page 1-1 looks at how FMLI works, describes the configuration of the screen when an FMLI application is running, and explains the function of each screen area. The concept of a frame is defined.
- “Programming with FMLI” on page 1-5 describes the three types of frames—menus, forms, and text frames—in which your application is presented, and some of the ways you can customize their appearance in the frame definition files you write. Application level files, in which you can define characteristics of your application as a whole, are also described.
- “An Example Application” on page 1-10 presents a simple example of an FMLI application and tells how to execute and exit from it.
- “Writing an Internationalized Application” on page 1-12 describes the guidelines for writing applications whose language-dependent output is not hard-coded in the frame definition files.
- “Using an FMLI Application” on page 1-13 describes how to work in menu, form, and text frames, navigate between frames, and execute commands.

## What Is FMLI?

The Form and Menu Language Interpreter provides a framework for developers to write applications and application interfaces that use menus and forms. It controls many aspects of screen management for you. That means you do not have to be concerned with the low-level details of creating or placing frames, providing users with a means of navigating between or within frames, or processing the use of forms and menus. Nor do you need to worry about what kind of terminal your application will run on. FMLI takes care of all that for you.

FMLI is a high-level programming tool having two main parts:

- The Form and Menu Language is a programming language for writing scripts that define how an application will be presented to users. The syntax of the Form and Menu Language is similar to that of the UNIX system shell programming language, and includes the following: variable setting and evaluation, built-in commands and functions, use of and escape from special characters, redirection of input and output, conditional statements, interrupt signal handling, and the ability to set various terminal attributes. The Form and Menu Language also includes sets of *descriptors* that are used to define or customize attributes of frames and other features of your application.

- The Form and Menu Language Interpreter, **`fmli(1)`**, is a command interpreter that sets up and controls the video display screen on a terminal, using instructions from your scripts to supplement FMLI's predefined screen control mechanisms. FMLI scripts can also invoke UNIX system executables, either in the background or in full screen mode. The Form and Menu Language Interpreter operates similarly to the UNIX command interpreter **`sh(1).`** At run time it parses the scripts you have written, giving you the advantages of quick prototyping and easy maintenance.

## Screen Layout

The following figure shows the configuration of the screen when an FMLI application is running.

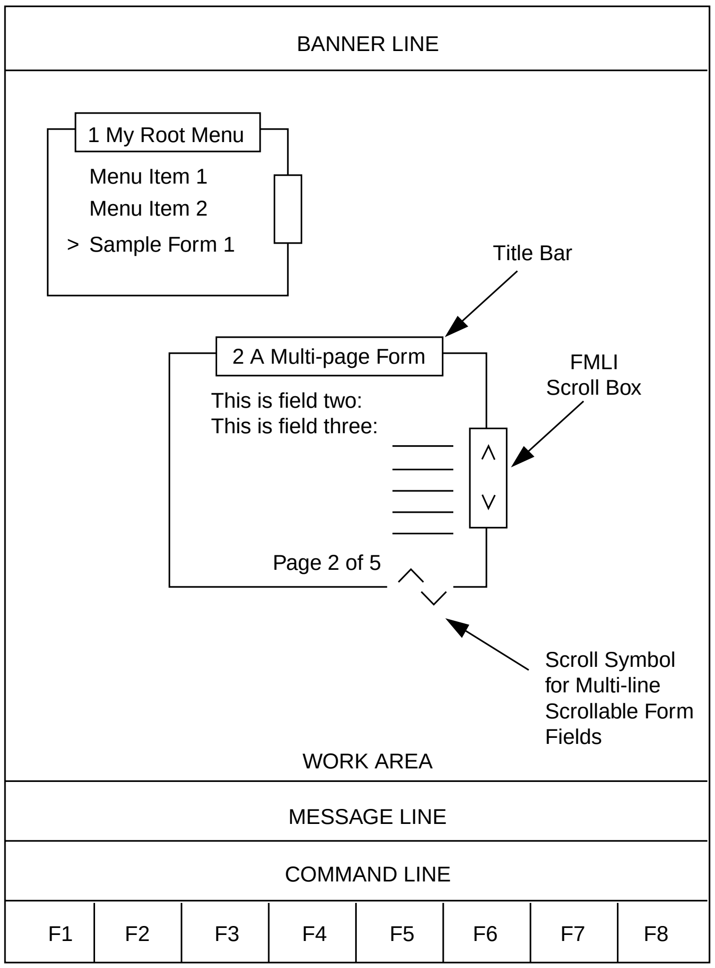


**Figure 1-1. The FMLI Screen**

FMLI divides the screen into the following five regions:

| | |
|---|---|
| Banner Line | The banner line displays a one-line banner on the top line of the screen. By default, it displays a `Working` icon when FMLI is busy. You can redefine the banner line in the initialization file. |
| Work Area | The work area is the section of the screen where frames are displayed. This area starts on the second line of the screen and stops on the third line from the bottom of the screen. |

**NOTE**

If a terminal supports hardware function keys, FMLI will use the last line of the screen to display function key labels. The developer should be aware of this since the size of the work area will be decreased by one line (that is, it will stop on the fourth line from the bottom) to make room for these labels.

Message Line

The message line is the second line from the bottom of the screen. Messages generated by FMLI, or which you generate in your scripts, are displayed here. By default, a message remains on the message line until the next key is pressed; you can define messages that will remain on display permanently or until other user actions occur, such as navigation to another frame.

Command Line

The command line is the next to last line on the screen. Users access it by pressing `CTRL-j` or `CTRL-f c`, at which time a --> prompt appears on the line. Any FMLI command, application-specific command, or UNIX system executable can be executed from the command line.

Screen Labels for Function Keys (SLKs)

The last line of the screen displays screen labels that correspond to the eight function keys found on many keyboards. Screen-labeled keys, or SLKs, allow users to invoke FMLI or application-specific commands easily by pressing one key. FMLI provides two sets of screen labels for the function keys. Your scripts control which set is displayed at any given time. FMLI predefines the SLKs in the first set, assigning each a default screen label and function depending on the type of frame current. (Figure 1-5 in “Using an FMLI Application” on page 1-13 shows the functions assigned by default to screen-labeled keys when a menu, form, or text frame is current.) The second set is not predefined—you can define this set specifically for your application. In the first set, you can rename or disable function keys `F1` through `F7` (but they cannot be redefined), and redefine function key `F8`. In the second set, you can define application-specific commands for function keys `F9` through `F16`. Keep in mind, though, that if you redefine key `F8` or `F16` to be something other than `CHG-KEYS`, your users will lose the ability to access the alternative set of function keys. A complete discussion of screen-labeled keys and how to define, disable, or redefine them is contained in Chapter 4.

Since some keyboards do not have function keys, FMLI predefines alternative keystroke sequences whose use is equivalent to that of function keys `F1` through `F8`. These sequences have the form `CTRL-f` *n*, where *n* is the number of the corresponding function key. The alternative keystroke sequence for `F3`, for example, is `CTRL-f 3`. That means the user must hold down `CTRL` while pressing `f`, then press `3`.

**NOTE**

FMLI downloads alternative keystroke sequences into the function keys of some terminals at the user's request. For a discussion, see Appendix B.

## Frames

A frame is an independently scrollable portion of the work area surrounded by a border. By default, the dimensions of a frame are determined by FMLI. Several frames may be open simultaneously in the work area but only one frame can be current at a time. Frames are positioned in the work area so that overlap of other frames is minimized.

The current frame is the frame a user is working in. It is distinguished from other frames in the work area by its border and all features in its border being displayed in full-bright video attribute; non-current frames are displayed in half-bright video. (On terminals that do not have the half-bright video attribute, non-current frames are displayed some other way, inverse video, for example.) The current frame may cover parts of other frames in the work area.

All menu, form, and text frames can display the following features:

| | |
|---|---|
| Title Bar | Each frame displays a title bar in its top border. The title bar contains a frame ID number assigned by FMLI, and the title of the frame—an FMLI default title or one you define. |
| Scroll Box | Each frame that contains three or more lines of information displays a scroll box in its right border. A scroll box can house both an up symbol (^) and a down symbol (v). These symbols indicate to the user that there is more information before (^) or after (v) the information currently displayed in the frame; only the up symbol (^) will appear in the scroll box when a user is viewing page two of a two-page form, for example. A scroll box will be blank when all the information in the frame is currently visible. |

The current item in a menu frame is indicated by a more-than sign (>) to its left. Depending on the terminal, the item may also be shown in inverse video.

Scroll symbols will appear in the lower right border of a form frame if a scrollable multi-line field is current. Although not shown in the figure, scroll symbols will appear to the right of a field if a scrollable single-line field is current. For a discussion of scrollable form fields, see “Form Frames” on page 1-7.

## Programming with FMLI

Typically, the scripts for an FMLI application include a set of frame definition files, each defining a single menu, form, or text frame. These files define the frames users will see when they execute your FMLI application, and the operations that can be done in frames. In addition, most FMLI applications include three (optional) application level definition files: an initialization file, a commands file, and an alias file. These files define global features of your application, such as the colors of various screen elements, application-specific commands, and aliases for path names that can make your code easier to read and maintain.

## Frame Definition Files

A frame definition file is a file made up of statements recognized by the **`fmli`** command interpreter. Three types of frames can be defined in FMLI: menu frames, form frames, and text frames. FMLI recognizes the type of frame you are defining based on the contents of the frame definition file and certain file naming conventions. The following sections briefly describe these three types of frames. Detailed explanations of how to write frame definition files can be found in Chapter 3.

### Menu Frames

A menu in FMLI is a method for displaying a list of selections in a frame, determining the user's selection, and taking action based on the selection. The title bar of a menu frame displays a default name for the menu (Menu) or one you define, and an identification number assigned to the frame by FMLI.

#### Single-column and Multi-column Menus

By default, FMLI presents menus with 10 or fewer items in a single left-justified column; if the number of items is greater than 10, FMLI attempts to create a multi-column menu with a 3:1 aspect ratio of width to height. You can explicitly define the number of rows and/or columns you want in a menu (see Appendix A for a table describing the way FMLI calculates rows and columns in menus). Menu items are presented in a single scrollable column if an entire menu cannot fit on the screen at once. Appropriate scroll symbols appear in the scroll box in the right-hand border of the frame.

#### Single-select and Multi-select Menus

You can define a menu to be either single-select or multi-select. In a single-select menu, the user can select only one item. When the user presses ENTER (the key or SLK) while the cursor is positioned on a menu item, the backquoted expression associated with the item is evaluated, and any FMLI command associated with the item is executed.

**NOTE**

Depending on the keyboard being used, the carriage-return key may be called ENTER, RETURN, or something else. Throughout this guide ENTER is used to represent the carriage-return key.

In a multi-select menu, the user can select more than one item. When the user presses MARK (the key or SLK) while the cursor is positioned on a menu item, the item is marked with an asterisk (*) to its left, and the backquoted expression associated with the item is evaluated. Any FMLI command associated with the item is ignored. When, after having marked all desired items, the user presses `ENTER`, the `done` descriptor is evaluated, and any FMLI commands defined by the descriptor are executed. Backquoted expressions, descriptors, and FMLI commands are discussed in Chapter 2.

## Form Frames

A form in FMLI is a method for displaying and prompting for information. To the user, a form looks like a fill-in-the-blanks questionnaire. The title bar of a form frame displays a default name for the form (`Form`) or one you define, and a frame ID number assigned to the frame by FMLI. A form comprises fields, which have two parts: a field label (the name of the field) and an area in which to enter a value for the field. You can define default field values that are displayed in the input area whenever the form is opened or updated.

### Multi-line and Scrollable Fields

You can define the field input area to be multi-line and/or scrollable. A scrollable form field allows users to enter more input in the field than its display area is sized for. If a scrollable multi-line field is current, appropriate vertical scroll symbols appear in the bottom right border of the frame: `^`, `v`, or both. For a scrollable single-line field, appropriate horizontal scroll symbols appear to the right of the display area: `<`, `>`, or `=` if there is more information before and after the information currently displayed.

### Multi-page Forms

A form can be more than one page long, in which case it can scroll a page at a time. The up and down symbols in the scroll box inform users that they are positioned on the first page, the last page, or one of the middle pages of a form. If you want to indicate the page more precisely, you can include a label such as `Page 2 of 5` in a form, as described under the `name` entry in the “Form Frame Descriptors” on page 3-26.

### Validating Field Values

In forms, you can use the field descriptor `valid` to validate the value a user enters in a field, or the descriptor `validOnDone` to validate the relationship between values of different fields (as when the validity of the value entered in field `x` depends on the value entered in field `y`), or both. In all cases, the user will not be able to save the form until the values pass the validation test.

### Choices Menu

A choices menu is a way to show users the valid choices for a field in a form. When you define a choices menu for a field, you can choose whether the user will toggle through the choices in the field itself, or whether the choices will be displayed in a pop-up menu. When a user selects a value from a pop-up choices menu, it is automatically entered in the field to which the menu applies.

### Text Frames

Text frames are primarily used to display read-only information, such as on-line help for the user. The title bar of a text frame displays a default name for the frame (`Text`) or one you define, and a frame ID number assigned to the frame by FMLI. The frame will be scrollable if all of its text will not fit in the display at one time; appropriate scroll symbols will appear in the scroll box in the right-hand border of the frame. If the text frame descriptor `edit` evaluates to TRUE, users will be able to change the text in the frame.

Text frames may be defined with text frame definition files, just as menus and forms are. Simple text frames may also be specified using a shorter notation, without a frame definition file, using the **`textframe`** command.

## Application Level Definition Files

Application level definition files define attributes of the application as a whole. There are three optional application level files. The initialization file and the commands file allow you to customize the appearance and functionality of your entire application; the alias file allows you to streamline references to source files in your code. The following sections briefly describe these three types of application level definition files. Detailed explanations of how to write them can be found in Chapter 4.

### Initialization File

An initialization file defines attributes of the application as a whole. You can define an introductory frame (such as a copyright notice), changes to the default banner line, the colors of various elements of the FMLI screen, whether users will be able to access the UNIX system directly from your application, and the names of and commands assigned to screen-labeled keys, among other things.

### Commands File

A commands file allows you to define new commands for users of your application, and redefine or disable existing FMLI commands. The new commands can be executed from the FMLI command line or the FMLI `Command Menu`, as described in the next section.

### Alias File

An alias file contains lines of the form *alias*=*pathname*. An alias can be assigned a single path to a file or device, or it can be assigned a series of paths to be searched (similar to the way `$PATH` is searched in the UNIX shell). Using aliases will make the code in your other definition files more readable.

## Terminal Independence

FMLI uses the UNIX System V **`terminfo`** database to determine the values of terminal-dependent capabilities. The default path to this database is **`/usr/share/lib/terminfo`** if the environment variable `TERMINFO` is not set. New terminals not described in this database can be added to the terminfo database under a subdirectory named by the first character in the terminal's name. For example, the 5425 terminal description would be in **`$TERMINFO/5/5425`**.

To ensure that the terminal is initialized properly for your FMLI application, include the command

```
tput init
```

in the executable or script that invokes your application. If you choose not to do that, the documentation for your application should remind users to place this command in their **`.profile`** file after the `TERM` variable is set and exported.

**NOTE**

Terminal attribute settings can be lost when a user returns to an FMLI application after having used a full-screen application executed via the FMLI **`run`** built-in utility. To prevent this from happening, the full-screen application can execute **`tput init`** before returning to FMLI.

FMLI will work on any asynchronous terminal that

- displays 80 characters across
- has at least 22 simultaneously visible lines
- has a proper **`terminfo`** entry in the host computer.

It may be possible to run FMLI on smaller screens if you define the size and position of frames to fit within the screen's limits. However, some elements of screen layout, such as the screen labels for the function keys, may be truncated.

FMLI downloads alternative keystroke sequences into the function keys of some terminals at the user's request. For a discussion, see Appendix B.

## Recovering after Abnormal Termination

In the case of an abnormal termination of an FMLI application, users can execute

```
CTRL-j stty sane CTRL-j
```

to restore the screen. Until this is done, user input might not be displayed on the screen, giving the appearance that the computer is hung or down. If the user executes this command to recover, it will also be necessary to execute

```
stty tab3
```

to ensure a sane screen. Borders of frames may be distorted otherwise.

## Internationalization Support

FMLI accepts as input any character from a standard 7- or 8-bit character set. This means that descriptor and variable values and application-specific command names may be coded in a language other than English, provided the language implementation employs a standard 8-bit code set. It also means that users may enter input in a form, or edit the text in a text frame, in any such language. Note, however, that the built-in utilities **`fmlexpr(1F)`**, **`fmlgrep(1F)`**, and **`regex(1F)`** do not support regular expression matching for non-ASCII character sets, and that FMLI error messages are always displayed in English.

FMLI uses the **`setlocale(3C)`** function to examine the user's environment for a current *locale*—a collection of information that describes conventions appropriate to some nationality, culture, and language. This information is stored in databases that describe how to sort or classify characters, for instance, according to these conventions. If such databases exist on a user's system, they are accessed through the `LANG` variable in the user's environment. An application coded for a German locale, then, should instruct users to set the `LANG` environment variable to `de`[utsche]; character classification, sorting, and so on will be done in the appropriate way. For details on this mechanism, see the **`setlocale(3C)`** manual page.

# An Example Application

Here is an example FMLI application consisting of a menu frame definition file named **`Menu.sample`** and a text frame definition file named **`Text.welcome`**:

```
menu=TOP MENU

name=date
action=`date | message`nop

name=welcome
action=open Text.welcome

name=exit
action=exit
```

**Figure 1-2. Menu.sample: A Simple Menu Frame Definition File**

The file **`Menu.sample`** is named according to the conventions defined for FMLI frame definition files (Chapter 2 contains a complete discussion of file naming conventions), and defines the initial frame to be opened when this example application is run.

The first line of code uses the FMLI descriptor `menu` to define the title that will appear in the title bar of the menu frame. Many FMLI descriptors have a default value that will be used if you do not explicitly define the descriptor in the frame definition file. If the `menu` descriptor were not defined in this file, the title of the menu would default to `Menu`.

The next two lines of code define the first menu item. The `name` descriptor defines the name of the item to be `date`. The `action` descriptor defines what will happen when the user selects the `date` menu item: the UNIX system **`date(1)`** command will be run and the output will be piped to the FMLI built-in utility **`message(1F),`** which displays the output of **`date`** on the FMLI message line. The FMLI command **`nop`** does nothing (no operation), but must be present because FMLI expects the `action` descriptor ultimately to evaluate to an FMLI command. If it doesn't, the terminal will beep.

The fourth and fifth lines of code define the menu item `welcome` and the action to execute when `welcome` is selected: open another frame in the work area, in this case the text frame `Text.welcome`, which displays a welcome message. Figure 1-2 shows the contents of the text frame definition file **`Text.welcome`**.

The last two lines of code define the menu item `exit` and its action: run the FMLI command **`exit`**, which terminates the FMLI session and returns the user to the UNIX shell.

```
title="WELCOME"
rows=3
text="Welcome to my application.
I hope you enjoy yourself
while you are using it."
```

**Figure 1-3. Text.welcome: A Text Frame Definition File**

The file **`Text.welcome`** defines a text frame and is named according to the conventions for text frame definition files. The `title` descriptor defines the title that will appear in the title bar of the text frame, in this case `WELCOME.` The `rows` descriptor defines the vertical size of the text frame in lines of text. The width of the frame in this example is determined by FMLI. The `text` descriptor defines the words that will be printed in the body of the text frame.

To execute this application, invoke **`fmli`** as follows:

```
fmli Menu.sample
```

In this example, **`Menu.sample`** is specified as the initial frame to open. An initial frame is a frame that is opened as an argument to **`fmli`** when it is invoked. There can be more than one initial frame. All initial frames remain displayed in the work area as long as the application is running; that is, a user cannot close or cancel an initial frame.

Users can exit from any FMLI application by pressing `CTRL-j` or `CTRL-f c` to access the FMLI command line and entering `exit.` Users of the example application could also select `exit` from `TOP MENU`.

In the next section, this example application will serve to illustrate how an FMLI application appears to users. If you want to supplement the discussion with a hands-on example, you can create copies of **`Menu.sample`** and **`Text.welcome`** in your file system and invoke **`fmli`** as shown above.

## Writing an Internationalized Application

Internationalized FMLI applications are applications whose language dependent output is not hard-coded in the frame definition files. The output (messages, menu items, frame titles, and so on) are encoded in a language-independent way in the definition file. At run-time, that is, when interpreting the respective file, FMLI retrieves the language-dependent output from a message catalogue which contains the output of the application in the language to which the system locale is set. If no message catalogue exists, FMLI tries to output a default message encoded in the frame definition file. If a default message does not exist either, you get the following message:

```
Message not found!
```

Writing an internationalized application requires that all strings that are to be presented on the screen must be described using the special syntax:

`"$$<`*catalogue_name*`>:<`*message_no*`.>:<`*default_message*`>"`

| | |
|---|---|
| *catalogue_name* | denotes the name of the catalogue in which the messages for a certain locale are stored. It can be stored in **`/usr/lib/locale/<locale>/LC_MESSAGES`** using the UNIX utility **`mkmsgs`**. |
| *message_no* | is the index to the respective message in the message catalogue. |
| *default_message* | is the message that is displayed if the locale is set to "`C`" or if no message catalogue exists in the current locale. |

The menu frame definition file shown in Figure would then look like this:

```
menu="$$uxmyapp:1:TOP MENU"

name="$$uxmyapp:2:date"
action=`date | message`nop

name="$$uxmyapp:3:welcome"
action=open Text.welcome

name="$$uxmyapp:4:exit"
action=exit
```

**Figure 1-4. Menu.sample: A Simple International Menu Definition File**

If you want your menu to be sorted automatically in any locale, you must indicate this by using the descriptor `autosort`. If it is set to `autosort=true`, the menu items will be presented in alphabetical order in any language. This would be independent from the order in the actual frame definition file.

The second menu item in Figure 1-3 would invoke an opening action on the file **`Text.welcome`**. It is stored in the current directory. If the application is to open a textfile that is translated to another language than the default, this textfile must be stored in the directory **`./$LANG/<file>`**. If the file **`Text.welcome`** was translated to German and if the application was to open that file, it would have to be stored in the directory

**`./De_DE.88591`** carrying the same name. FMLI automatically checks the locale before opening a textfile.

## Using an FMLI Application

This section discusses the *look and feel* of an FMLI application. It covers the way menus, forms, and text frames and other visual elements of the FMLI screen environment are presented to users (the *look*), the basics of navigation, the ways commands can be executed, the functions assigned by default to named keyboard keys, and how to use alternative keystroke sequences in the event named keys do not work, or exist, on a keyboard (the *feel*).

This information is important for two reasons. First, it describes the features that “come for free” with an FMLI application and around which you can design your own application. Second, you will need to include at least some of this information in your user documents. We can't help you describe what's in your menus, forms, and text frames, but we can help you describe the tasks that should be common to any FMLI application: working in menus, forms, and text frames, navigating between frames, executing commands, and getting help.

When you execute the command **`fmli Menu.sample`**, given the FMLI scripts just discussed, the terminal screen will look like this:

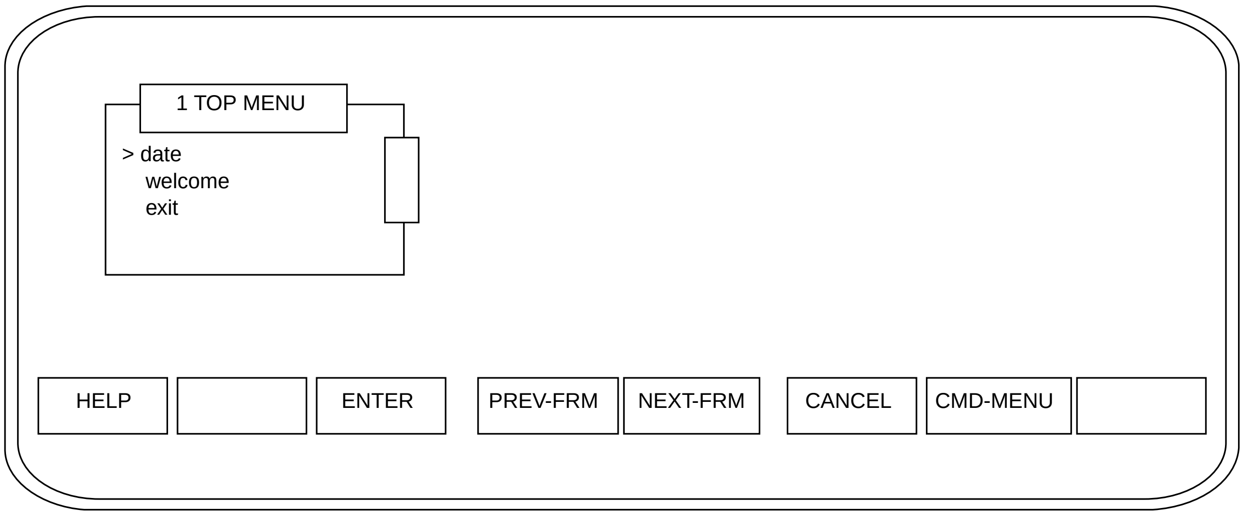


**Figure 1-5. Menu.sample: Screen Output**

This application illustrates most of the aspects of screen and frame style enforced by FMLI. (Screen and frame style refers to the way these elements are presented to users when an FMLI application is running.)

Since the frame opened initially in this example is a menu frame, the frame itself and the screen labels on the last line of the screen show the default appearance when a menu frame is current. The banner line (top line of the screen) is currently blank, but while this application was being loaded you may have noticed that the word `Working` appeared at the right on the banner line. The banner line is blank by default, except for the `Working` indicator. But you can define the banner line to display more information, and you can change the `Working` indicator. (See Chapter 4 and the **`indicator(1F)`** manual page for more information.)

The menu frame defined in **`Menu.sample`** and named on the **`fmli`** invocation line as the initial frame to open is displayed in the work area. The title bar of the frame displays the menu title and a frame ID number assigned automatically by FMLI. Each item defined in **`Menu.sample`** is listed in the order in which it was defined; the `>` symbol shows where the cursor is currently positioned. A scroll bar appears in the right-hand frame border because there are three items in this menu, although no scrolling symbols are shown because the entire menu can fit in the frame at one time.

The two lines immediately above the function-key screen labels are the message line and the command line, although nothing is displayed on them at this point. You can press CTRL-j or CTRL-f c to navigate to the command line. Press ENTER or CTRL-j again to leave the command line without executing a command.

The last line of the screen displays the default set of screen labels for function keys when the current frame is a menu. The defaults are different when a form frame or text frame is current. Figure 1-5 shows the functions assigned by default to screen-labeled keys when a menu, form, or text frame is current. A complete discussion of screen-labeled keys and how to define, disable, or redefine them is contained in Chapter 4.

The features discussed in the following sections can be used in any FMLI application. Where appropriate, they will be explained in the context of the example application just discussed.

**Table 1-1. Default Screen-labeled Keys**

| Function Key | Menu Frame | Form Frame | Text Frame | Choices Menu | Command Menu |
|---|---|---|---|---|---|
| F1 | help | help | help | | help |
| F2 | mark* | choices | prevpage | | |
| F3 | enter | save | nextpage | enter | |
| F4 | prev-frm | prev-frm | prev-frm | | |
| F5 | next-frm | next-frm | next-frm | | |
| F6 | cancel | cancel | cancel | cancel | cancel |
| F7 | cmd-menu | cmd-menu | cmd-menu | | |
| F8 | chg-keys** | chg-keys** | chg-keys** | chg-keys** | chg-keys** |
| F16 | chg-keys** | chg-keys** | chg-keys** | chg-keys** | chg-keys** |

* Function key F2 is assigned the **`mark`** command only in multi-select menus. In single select menus F2 has no default assigned.

** Function keys F8 and F16 will default to `chg-keys` only if any of keys F9 through F15 are defined by the developer.

## Named Keys and Alternative Keystroke Sequences

Named keys are the keys on terminal keyboards that do something other than print an alphanumeric or special character. Named keys include `ENTER`, `TAB`, `DEL`, the function keys `F1` through `F8`, and although not strictly named, the arrow keys `<Down-Arrow>`, `<Up-Arrow>`, `<Right-Arrow>`, and `<Left-Arrow>`. Since many terminal keyboards will not have a complete set of named keys, FMLI predefines alternative keystroke sequences whose use is equivalent to named keys. The alternative keystroke sequence for `<Down-Arrow>` for example, is `CTRL-d`. That means the user must hold down `CTRL` while pressing `d`.

Some of the named keys are reserved for navigation and/or editing during an FMLI session. Navigation keys are named keys that, when pressed, cause the cursor to move. The default action assigned to a navigation key changes depending on whether you are in a menu, form, or text frame. For example, the named key `BEG` or the alternative keystroke sequence `CTRL-b` work as follows in the three types of frames:

| | |
|---|---|
| `menu` | moves the cursor to the first item in the menu, whether it is currently visible or not |
| `form` | moves the cursor to the first field of the current page of the form |
| `text` | causes the first frame full of text to be displayed |

The default action assigned to `BEG` in these three cases has a common element—moving to the beginning—but the meaning varies according to what kinds of things users need to do in each type of frame. A complete table of named keys recognized by FMLI (using **`terminfo`**) is provided in Appendix B and summarizes the action that will occur when these keys or their alternative keystroke sequences are pressed in menus, forms, and text frames.

## Navigating in a Menu

There are two methods of navigating in a menu frame. One is to use a navigation key. As an example of how navigation keys work, you can try the menu navigation keys in the example menu `TOP MENU`. The following list shows some of the keys you can use to navigate in a menu:

- `<Right-Arrow>` or the alternative keystroke sequence `CTRL-r` moves the cursor right one item in a multi-column menu, or down one item in a single-column menu. In a multi-column menu, it does not wrap. In a single-column menu, it wraps to the top of the column.
- `<Left-Arrow>` or the alternative keystroke sequence `CTRL-l` moves the cursor left one item in a multi-column menu, or up one item in a single-column menu. In a multi-column menu, it does not wrap. In a single-column menu, it wraps to the bottom of the column.
- `<Down-Arrow>` or the alternative keystroke sequence `CTRL-d` moves the cursor down one item, wrapping to the top of the column in a single-column menu, and the top of the next column in a multi-column menu. On

the last item in the last column of a multi-column menu, it wraps to the top of the first column.

- <Up-Arrow> or the alternative keystroke sequence CTRL-u moves the cursor up one item, wrapping to the bottom of the column in a single-column menu, and the bottom of the previous column in a multi-column menu. On the first item in the first column of a multi-column menu, it wraps to the bottom of the last column.

As you navigate in the menu, the > symbol shows which menu item is current. In a scrollable (by definition, single-column) menu, pressing <Right-Arrow> or <Down-Arrow> when the cursor is on the last item of the display will roll the contents of the menu up one line; pressing <Left-Arrow> and <Up-Arrow> when the cursor is on the first line of the display will roll the contents of the menu down one line. Note that pressing the named keys SCROLL-UP or SCROLL-DOWN will roll the contents of a scrollable menu up or down one line, respectively, without moving the cursor.

The other method of navigating in a menu frame is to type the name of the item to which you want to move. You don't have to type the full name, or worry about upper and lower case. When you type a character, the cursor moves to the first item in the menu that matches the string typed so far. If you type the letter w, for example, the cursor moves to the first menu item that starts with w or W. If you then type r, the cursor moves to the first item that starts with the letters wr. When a string cannot be matched, the terminal bell sounds, or the screen flashes, depending on the terminal, and an error message is displayed on the message line. The cursor wraps around when it reaches either end of the menu. In a scrollable menu the display scrolls as necessary.

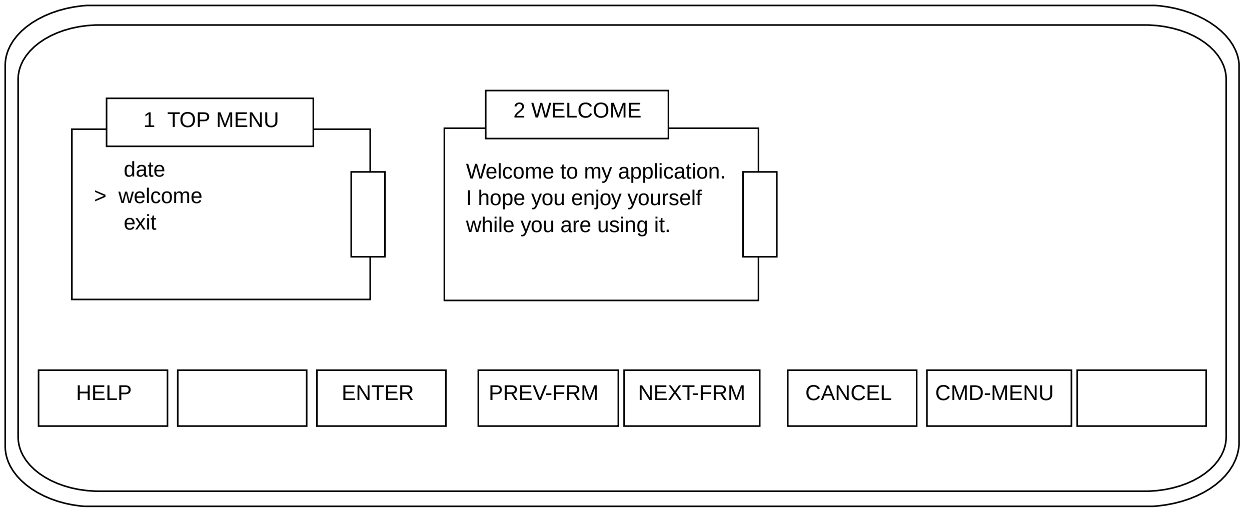
**NOTE**

If you start to type the name of a menu item and the cursor moves, and you then decide to select something else, you must use BACKSPACE to erase the characters already typed, or press one of the navigation keys before character matching can be used again.

## Selecting Menu Items

As noted, you can define a menu to be either single-select or multi-select. In a single-select menu you can select only one item; in a multi-select menu you can select more than one item. To select an item in a single-select menu, press ENTER (the key or SLK) while the cursor is positioned on the item. To select items in a multi-select menu, first mark each of the desired items by pressing MARK (the key or SLK) while the cursor is positioned on the item; an asterisk (*) will appear to the left of the item. Now press ENTER to select the marked items.

If you are running the example application, you can see how a single-select menu works by navigating to the item welcome and pressing ENTER. The screen will look like this:



Welcome to my application.
I hope you enjoy yourself
while you are using it.

**Figure 1-6. Menu.sample: Screen Output after Selecting welcome**

Notice that the text frame defined in **`Text.welcome`** is displayed in the work area, its frame ID number is `2`, and the screen labels shown on the last line of the screen now display the default labels for text frames.

## Navigating in a Form

When a form is opened, the cursor is placed in the first character position in the first field of the form. The following list shows some of the keys you can use to navigate in a form:

- In a single-line field, `ENTER` or the alternative keystroke sequence `CTRL-m` moves the cursor to the next field, whether it is below the current field or to the right, wrapping from the last field of the form to the first. In a multi-line field, it moves the cursor to the next line, scrolling on the last line if the field is scrollable, stopping and beeping if it is not. That is, you cannot use this key to navigate from a multi-line field.
- In a single- or multi-line field, `TAB` or the alternative keystroke sequence `CTRL-i` moves the cursor to the next field, whether it is below the current field or to the right, wrapping from the last field of the form to the first.
- In a single- or multi-line field, `BACKTAB` or the alternative keystroke sequence `CTRL-t` moves the cursor to the previous field, whether it is above the current field or to the left, wrapping from the first field of the form to the last.
- In a single-line field, `<Down-Arrow>` or the alternative keystroke sequence `CTRL-d` moves the cursor to the next field below the current field, wrapping from the last field of the column to the first. In a multi-line field, it moves the cursor to the next line; on the last line of the field, it moves the cursor to the next field below the current one.

- In a single-line field, <Up-Arrow> or the alternative keystroke sequence CTRL-u moves the cursor to the previous field above the current field, wrapping from the first field of the column to the last. In a multi-line field, it moves the cursor to the previous line; on the first line of the field, it moves the cursor to the previous field above the current one.
- <Right-Arrow> or the alternative keystroke sequence CTRL-r moves the cursor non-destructively one character to the right in a field. It does not wrap to the next field, or the next line in a multi-line field.
- <Left-Arrow> or the alternative keystroke sequence CTRL-l moves the cursor non-destructively one character to the left in a field. It does not wrap to the previous field, or the previous line in a multi-line field.

Multi-page forms and scrollable single- and multi-line fields scroll as necessary.

**NOTE**

Generally speaking, if a user enters invalid data in a field that has an associated validation test, navigation away from the field is not permitted. With this release, however, if no data have been entered or modified in the field since it became current, validation only occurs if the user attempts to use the ENTER key to leave the field. A user can leave the field with some other navigation key (such as one of the arrow keys). In such cases, validation of the field is delayed until the SAVE key is pressed. Thus, a user can use a key other than ENTER to navigate in a form when, say, leaving the current field blank would cause it to fail a validation test. You may want to point out this change in behavior to users who are familiar with the old behavior.

## Editing and Saving a Form

When a field is current, you are automatically in overtype mode: the character you type replaces the character under the cursor. If, immediately after navigating to a field, or attempting to navigate from a field with an invalid value, you type over the first character in the field, the rest of the line is automatically cleared. You then enter characters as you would if the field were blank. You save the values you have entered in fields and close the form by pressing the SAVE SLK.

Here are some of the named keys you can use to edit a form field:

- DEL, DELETE-CHAR, or the alternative keystroke sequence CTRL-x deletes the character under the cursor and closes the gap.
- INSERT-CHAR or the alternative keystroke sequence CTRL-a inserts to the left of the character under the cursor the next single character entered.
- CLEAR, CLEAR-LINE, or the alternative keystroke sequence CTRL-y clears the current line.
- RESET or the alternative keystroke sequence CTRL-f r restores the default value of a field.

## Using a Choices Menu

You access a choices menu and toggle through choices in the field itself by pressing the CHOICES SLK. You can use named keys to navigate and select items in a pop-up choices menu as you would in any other menu. As noted, when you select a value from a pop-up choices menu, it is automatically entered in the field to which the menu applies. Note that if you navigate away from a pop-up choices menu it disappears immediately.

## Navigating in and Editing a Text Frame

Here are some of the navigation keys you can use in a text frame:

- <Right-Arrow> or the alternative keystroke sequence CTRL-r moves the cursor non-destructively one character to the right.
- <Left-Arrow> or the alternative keystroke sequence CTRL-l moves the cursor non-destructively one character to the left.
- <Down-Arrow> or the alternative keystroke sequence CTRL-d moves the cursor down one line.
- <Up-Arrow> or the alternative keystroke sequence CTRL-u moves the cursor up one line.

The cursor does not wrap when you use these keys in text frames. Scrollable text frames scroll as necessary. Pressing the named keys SCROLL-UP or SCROLL-DOWN will roll the contents of a scrollable text frame up or down one line, respectively, without moving the cursor. Pressing the screen-labeled keys PREVPAGE or NEXTPAGE will move the cursor to the first character of the previous page or the first character of the next page, respectively, in scrollable text frames.

Except for RESET, the keys described in “Editing and Saving a Form” on page 1-18 work the same way in editable text frames.

## Navigating between Frames

Navigation between frames comprises simple moves and command actions that change which frame is current. The following list describes all the ways to move between frames:

- The PREV-FRM and NEXT-FRM SLKs are assigned the **`prev-frm`** and **`next-frm`** FMLI commands, respectively. Pressing either of these SLKs will cause you to navigate from frame to frame. The frame navigated to becomes the current frame on the screen, and the frame navigated from becomes non-current. FMLI keeps a list of each frame that has been the current frame. For example, if frame *N* is current, pressing PREV-FRM will cause you to navigate to the frame that was current when frame *N* was opened. Pressing NEXT-FRM while in frame *N* will cause you to navigate to the last frame opened while frame *N* was current. Since the PREV-FRM and NEXT-FRM commands are always relative to the current frame, the order in which they cause navigation through the frames displayed in the work area does not always follow frame ID order, and wrapping occurs in the order described above.

- Selecting **`frm-mgmt`** from the `Command Menu` will bring up a choices menu that includes the item `LIST`. Selecting `LIST` brings up another choices menu listing all opened frames. When you select a listed frame, the choices menu disappears and the selected frame becomes current. If you do not want to select a listed frame, you can press the `CANCEL` SLK. The choices menu will disappear and you will be put back in the frame you started out in.
- You can enter a frame ID number on the command line and press `ENTER`. The frame with that ID number will become current. This is equivalent to executing the **`goto`** command on the command line with a frame ID number as an argument, or selecting **`goto`** from the `COMMAND MENU`, entering a frame ID number after the prompt on the command line, and pressing `ENTER`. For a discussion, see the next section.
- Opening (selecting) a frame will always cause navigation to that frame.
- Closing (canceling) a frame will cause navigation to the *backup frame*. The concept of a backup frame allows FMLI to maintain a linked list of frames so that users always have a frame to which they will automatically be returned when a frame is canceled. The logic behind what frame is backup frame for another is not always immediately apparent to users, but its purpose is to maintain a linked list in which every frame is the backup frame for only one other frame. Usually, the backup frame is the frame that was current when the frame being closed was opened. However, if a user has been doing rather convoluted navigation, a frame's backup frame can change dynamically. For example if frame A is current and frame B is opened, A becomes B's backup. If A was already C's backup, B becomes C's backup.
- The **`cleanup`** command will close all frames whose `lifetime` descriptor evaluates to `shortterm` or `longterm.` If the `lifetime` descriptor evaluates to `immortal,` it means the frame cannot be closed except by exiting from the FMLI application. Note that the `lifetime` descriptor for initial frames, that is, frames opened as arguments to **`fmli`** when it is invoked, evaluates to `immortal` by default and cannot be redefined.

## Executing Commands

This section describes how users execute FMLI commands. A complete discussion of FMLI commands can be found in Chapter 2.

### The Command Menu

The `Command Menu` is an FMLI-supplied menu that lists a subset of FMLI commands. By default, function key `F7` is labeled `cmd-menu` when a menu (except the

Command Menu itself), form, or text frame is current. Pressing the cmd-menu SLK causes the Command Menu to appear in the work area:

```
          Command Menu
> cancel          next-frm
  cleanup         prev-frm
  exit            refresh
  frm-mgmt        unix-system
  goto            update
   help
```

The Command Menu will reflect the contents of the commands file. That is, if you rename, redefine, or disable an existing FMLI command in the commands file, or if you define a new command for your application, it will be added to or removed from the Command Menu as appropriate. You execute a command from the Command Menu by selecting it just as you would select an item in any other menu.

## The Command Line

You access the FMLI command line by pressing CTRL-j or CTRL-f c. At the `-->` prompt, you enter the name and arguments, if any, of the FMLI command you want to execute and press ENTER. All FMLI commands can be executed from the command line except those you have disabled in the commands file.

If a message catalogue for a certain language exists in your system, the commands in the command menu will be presented in the respective language. Commands in the command line must always be entered in English even though they might be output in another language in the command menu.

If what you enter on the command line is not a known FMLI command, it is interpreted according to the following default behavior. If what you enter is an integer, the command that will be executed defaults to **`goto`** *integer*; if you enter `2`, navigation to frame 2 will occur. If what you enter is anything other than an integer or a known FMLI command, the command that will be executed defaults to **`open`** *what-is-entered.*

You can try using the command line now if you are running the example application. Press CTRL-j, enter the command **`release,`** and press ENTER. This FMLI command returns the release number of the version of FMLI you are currently running. The words `FMLI Release 4.2 P`*n*, where *n* is the version number, should be displayed on the message line.

If the Command Menu is the current frame when you press CTRL-j or CTRL-f c, the Command Menu will disappear and the command the cursor is currently positioned on will appear on the command line after the prompt.

**NOTE**

In releases previous to FMLI 4.0, `CTRL-z` was used to access the command line. In UNIX System V Release 4.0, `CTRL-z` is used for job control in **`jsh`** or **`ksh.`** For that reason, using `CTRL-z` will suspend an FMLI application. To resume the FMLI application, use the **`fg`** command. (See the **`sh(1)`** manual page for detailed information on job control.)

## Screen-labeled Function Keys

Pressing a screen-labeled function key (SLK) results in the execution of the command defined for that function key. Many of the default FMLI commands shown on the SLKs can also be selected from the `Command Menu` or executed from the command line. (Not all FMLI commands shown by default on the SLKs appear in the `Command Menu`, and vice versa, but all FMLI commands can be executed from the command line.) If you have defined a SLK (either in an initialization file or in a frame definition file) to execute an application-specific command or a different FMLI command from the default, then pressing that SLK will execute the command you have defined for that key.

FMLI provides for international SLKs. User-defined SLKs in initialization or frame definition files can be internationalized with the following syntax:

`$$`<*catalogue_name*>`:`<*message_no*>`:`<*default_message*>

See "Writing an Internationalized Application" on page 1-12 for more details.

## On-line Help

When a menu, form, or text frame is current, function key `F1` is labeled `HELP` and assigned the FMLI **`help`** command by default. Pressing the `HELP` SLK or selecting the **`help`** command in the `Command Menu` results in the FMLI `help` descriptor being evaluated. Typically, you use the `help` descriptor to open a text frame that presents information on the use of the frame or command. This can be done using a text frame definition file, or more simply with the **`textframe`** command. But you can define `help` to be anything, such as a message to be printed on the message line or a UNIX system executable.

On-line help is available for each FMLI command. You can request help on a command by pressing `CMD-MENU` to access the `Command Menu`, navigating to the command for which you want help, and pressing `HELP`. You can do the same thing by entering a command of the form

**`help`** *command_name*

on the command line, where *command_name* is the name of the command for which you want help.

The on-line help information can be output in languages other than English. To do this, the FMLI message catalogue must be translated.

There are other ways you can provide users with help on the use of your application. You can define a short descriptive tag to be displayed alongside an item in a menu as a "memory jogger" on the use of that item. The choices menu that you can define for a field in a form frame can be considered a kind of help. Chapter 3 presents examples of these and other ways to provide users with on-line help. Help can also be provided via any of the FMLI descriptors that display a message on the message line, such as `choicemsg`, `itemmsg`, or `fieldmsg`. The built-in utility **`message(1F)`** can also be used to display information on the message line.

The on-line help can be encoded with the `$$`-syntax so that output in different languages is possible. If a text file needs to be read (when using the help descriptor and the **`readfile`** command), its different language versions should be stored under **<***dirname***>/LANG/<***file***>**. The default version should be stored in **<***dirname***>/<***file***>**.

## Accessing the UNIX System

You can access the UNIX system by selecting the FMLI **`unix-system`** command from the `Command Menu` or by entering the **`command_name`** on the command line. When you invoke **`unix-system`**, the FMLI screen clears and you are put in a full-screen UNIX shell. When you exit from the UNIX system, a prompt message appears requesting that you press `ENTER` to continue. The FMLI screen returns in the same condition it was in before the **`unix-system`** command was issued. You can control user access to the UNIX shell by disabling the **`unix-system`** command in the commands file (see "The Commands File" on page 4-12 for a discussion of how to disable FMLI commands).

By default, you can run UNIX system commands from the FMLI command line by prefixing an exclamation mark (`!`) to the command. (Whitespace is ignored before `!` when it is used as a UNIX system escape on the command line.) The **`run(1F)`** built-in function can also be used to execute UNIX system commands from the FMLI command line (see the **`run(1F)`** manual page for details). You can use the `nobang` descriptor to disable these features, as described in "The Initialization File" on page 4-1.

# 2
# The Form and Menu Language

# 2 The Form and Menu Language

## Introduction

The Form and Menu Language is a high-level "shell-like" language for defining menus, forms, and text frames for your application. A menu, form, or text frame definition is stored in a *frame definition file* made up of statements recognized by the Form and Menu Language Interpreter—the **`fmli`** command. Frame definition files can contain fixed descriptions of the contents of the frame and/or code that will dynamically generate the contents. When **`fmli`** is invoked for the scripts you have written, the frame definition files are parsed, and the frames that will be displayed on the screen are generated.

This chapter summarizes the syntax of the various elements of the Form and Menu Language.

## Syntax, Rules, and Conventions

The following sections discuss the general rules and the conventions which apply to the Form and Menu Language. Specifics of syntax for particular elements of the language are covered in the appropriate sections.

### Naming Conventions for Frame Definition Files

On the **`fmli`** command line, frame definition files are recognized as arguments only when they are named in accordance with the following conventions:

- **`Menu.`***name* is the format for names of menu definition files
- **`Form.`***name* is the format for names of form definition files
- **`Text.`***name* is the format for names of text definition files

where *name* can be any string that conforms to the UNIX system file naming conventions.

In a frame definition file, however, file name arguments to the **open** command can follow the above conventions, or they can have any valid UNIX system file name as long as one of the type casts `MENU`, `FORM`, or `TEXT` is used to identify the kind of frame definition file being opened (see "Type Casts" on page 2-2 below for more information).

## Comments

Comments can be included by beginning a line with the pound sign character. A pound sign, #, when it is the first non-whitespace character on a line, causes all following text up to a newline to be ignored by FMLI. (Inside single quotes (`' '`), double quotes (“ ”), and backquotes (`` ` ` ``), the pound sign has no special meaning. Thus, comments cannot be included in backquoted expressions.)

## Case Sensitivity

Some elements of the Form and Menu Language are case-insensitive, and some are case-sensitive.

The case-insensitive elements are

- descriptor names
- FMLI command names
- type casts
- descriptor values of type Boolean

The case-sensitive elements are

- arguments to commands
- names of frame definition files
- descriptor values of type string
- variable names
- FMLI built-in utility names and UNIX executable names, within back-quoted expressions

## Type Casts

### File Type Casts

A file type cast is an identifier that indicates to FMLI the type of frame definition file being opened when the file's name does not follow the naming conventions for frame definition files. There are three type casts for frame definition files: `MENU`, `FORM`, and `TEXT`. Each can be used as the first argument to the **open** command. For example,

```
action=open form user.address
```

identifies **`user.address`** as a form definition file. A frame definition file can be identified by using both the file naming convention and a type cast, although only one or the other is required.

Note that file type casts cannot be used to identify the initial frame(s) to open when **`fmli`** is invoked.

### Type Casts That Change the Time of Descriptor Evaluation

By default, FMLI determines how often descriptors are evaluated. You can use the `const` type cast to make sure that a descriptor is evaluated only once, no matter how many times it is referenced, or the `vary` type cast to make sure that a descriptor is evaluated whenever it is referenced. In either case, the cast must appear immediately after the equal sign (`=`) on the descriptor line, as in the following:

```
    show=const `set -l DAY=date +%a; test “$DAY” = “Friday”`
```

We'll explain what this example does, and more generally, why you might want to use `const` and `vary` in “Descriptor Evaluation” on page 2-10.

## Special Characters

Special characters in FMLI scripts are:

| | | | |
|---|---|---|---|
| double quote | “ | right-angle bracket | > |
| single quote | ' | left-angle bracket | < |
| backslash | \ | <newline> | |
| backquote | ` | <space> | |
| dollar sign | $ | <tab> | |
| vertical bar | \| | left brace | { |
| ampersand | & | right brace (in the `rmenu` descriptor only) | } |
| semicolon | ; | integer 2 (when followed by >) | 2 |
| pound sign (in the first non-whitespace column of a line) | # | | |
| dollar dollar | $$ | | |

Some FMLI built-in utilities, such as **`regex`** and **`fmlgrep`**, have other special characters. These are discussed with the appropriate utilities in the section 1F manual pages.

## Quoting Mechanisms

FMLI supports quoting mechanisms, similar to those used in the UNIX system shell, for disabling the meaning of special characters in a string. Each quoting mechanism has a different function, as defined below.

- Backslash (`\`): A backslash causes the next single character to be taken literally. That is, any special meaning of the character following a backslash is turned off. (In some cases, multiple backslashes may be required to escape the special meaning of a character.)
- Single quotes (`' '`): Any string inside of single quotes is taken literally and as a unit. Inside single quotes, only the backslash (`\`) has special meaning.
- Double quotes (`" "`): Double quotes group the text between them as a unit, but still allow variable expansion and the use of backquotes. Inside double quotes, only backslash (`\`), backquote (`` ` ``), and dollar sign (`$`) retain their special meanings. Carriage returns inside double quotes are enforced.
- Backquotes (`` ` ` ``): (Backquoted expressions are discussed in detail in the next section.) Any statement or series of statements may be enclosed in backquotes with the result that such a backquoted expression evaluates to the output of the last statement. Statements may be UNIX system executables or FMLI built-in utilities. Backquotes cannot be nested, except as provided for in **`regex`**. (See the **`regex(1F)`** manual page.)

**NOTE**

If a statement run in a backquoted expression changes the **`stty(1)`** setting, the FMLI session may be corrupted. Frames may not display correctly and the FMLI command line may not function (the latter occurs if RETURN is mapped to LINEFEED or to LINEFEED RETURN).

## Backquoted Expressions

Backquoted expressions may be coded as the value of a descriptor. They are evaluated at the time the descriptor is evaluated. When a backquoted expression produces output, it is considered part of the descriptor. This output must not produce an illegal value on the descriptor line. For instance, if the variable `MYVAR` is set to `hello`, then

```
action=`echo $MYVAR`open menu Mymenu
```

will be equivalent to

```
action=helloopen menu Mymenu
```

This produces an illegal descriptor value since `helloopen` is not a known FMLI command, and descriptors of type command must evaluate to a known FMLI command. As a result, the terminal will beep.

In addition to using backquoted expressions on descriptor lines, you can code them as "stand-alone" lines anywhere in menu, form, or text frame definition files. A stand-alone backquoted expression is one that starts a line, and it is evaluated when the frame definition file is opened, reread, or updated; before any descriptors are evaluated. Thus, if a stand-alone backquoted expression produces output to the message line, the output will appear before the frame being parsed is posted.

It is important to note that information can be passed to or from UNIX system executables and FMLI built-in commands using backquoted expressions. For example a menu item with the following definition of the `action` descriptor

```
action=`date | message` nop
```

passes the output of the UNIX system **`date`** command to the FMLI built-in utility **`message`**, which displays it on the message line.

Using this feature of the Form and Menu Language, you can generate the entire contents of a frame dynamically at run time. For an example of a menu generated this way, see the **`regex(1F)`** manual page and "Creating a Dynamic Menu" on page 3-23.

**NOTE**

In backquoted expressions, executables that expect standard input must be run via the **`run`** built-in utility. For example, if a user selects a menu item which has its `action` descriptor coded as ``action=`vi myfile`nop``, the FMLI session will appear to hang. The same action, coded as ``action=`run vi myfile`nop``, executes properly.

## Expression Operators

Several statements, utilizing FMLI built-in utilities or UNIX system executables, may appear inside a single backquoted expression, separated by one of the following operators:

- Semicolon (`;`): Statements separated by a semicolon are executed sequentially.
- Pipe ( `|` ): When statements are separated by a pipe symbol, the output of the first statement becomes the input to the second.
- AND (&&): The meaning of *statement1* && *statement2* is run *statement1* and if it succeeds, then run *statement2*.
- OR ( `||` ): The meaning of *statement1* `||` *statement2* is run *statement1* and if it fails, run *statement2*.

**NOTE**

FMLI does not allow statement grouping by using parentheses, such as can be done in the UNIX shell, namely, *statement1* && (*statement2*; *statement3*).

## File Redirection

The input of a statement may be redirected from a file by using `<` *file*. Similarly, the output of a statement may be sent to a file by using `>` *file*, or by using `>>` *file* to append output to the end of a file.

The output from standard error may be redirected by using `2>` *file* to send it to a file, or by using `2>>` *file* to append it to the end of a file.

As in the UNIX shell, whitespace between `>` or `<` and *file* is optional.

## Syntax Errors

In general, FMLI does not generate messages on syntax errors. Anything it doesn't understand is ignored. However, some of the built-in utilities such as **`fmlgrep`,** and the FMLI conditional statement, generate their own syntax error messages. For example, a misspelled descriptor will be ignored, but a syntax error in a use of **`fmlgrep`** may cause an error message. In addition, correctly coded descriptors will be ignored if they are used in the wrong context. For example, the `selected` descriptor will be ignored in a single-select menu (because `selected` has no meaning in that context).

When creating a new form, menu, or text frame, all quotes and backquotes must match. Quoting mismatches may cause the frame not to appear or appear incorrectly, or cause an **`fmli`** session to terminate (exit). Quoting mismatches are not reported.

## Variables

The Form and Menu Language Interpreter recognizes user-defined variables, and a set of read-only special FMLI variables (known as built-in variables), as well as UNIX shell variables such as `HOME` or `MAIL`.

Variables in FMLI are global. That is, variables defined in one frame or application level definition file are exported to all other frame definition files or application level files after the **`set`** command has been executed. If a frame is not opened during the execution of an FMLI session, variables defined in it are not available.

## User-defined Variables

User-defined variables are names to which you may assign string values using the FMLI built-in utility **`set`** (see the **`set(1F)`** manual page for complete details on its use). You can assign values to variables in the local environment, available to the current FMLI session only:

**`set -l`** *name*=*value*

**NOTE**

Local variables are available to all frames of your application, not just the frame in which they are set.

Or a variable can be made available to any application/process by placing the variable in a file, thus allowing another application (for example, another FMLI application) to retrieve the information:

**`set -f`***filename name***`=`***value*

where *name* is a sequence of letters, digits, and underscores that begins with a letter or an underscore, *value* is a string, and *filename* is the path name to a file that contains lines of the form *name*=*value*. If it does not already exist, *filename* will be created. Note that no spaces surround the equal sign (`=`).

The built-in utility **`set`** can also be used to set variables in the UNIX shell environment and export them to the current session and to its child processes:

**`set -e`** *name***`=`***value*

The built-in utility **`unset`** can be used to remove a variable assignment (see the **`set(1F)`** manual page for complete details on its use).

## Built-in Variables

The built-in variables are a set of special, read-only variables that are predefined in the Form and Menu Language. These built-in variables can only be referenced, but never set, in frame definition files. The built-in variables are as follows:

| | |
|---|---|
| `ARG`*n* | This variable evaluates to the *n*th argument passed to the corresponding form, menu, or text frame. |
| `DISPLAYH` | This variable evaluates to the height of the available frame display area, minus the three lines reserved for the message line, the command line, and the screen labels for function keys. `DISPLAYH` is placed in the UNIX shell environment. |
| `DISPLAYW` | This variable evaluates to the width of the available frame display area of the screen. `DISPLAYW` is placed in the UNIX shell environment. |
| `Form_Choice` | This variable evaluates to the last choice made from a choices menu. |
| `F`*n* | This variable evaluates to the current value of the *n*th field. |

**NOTE**

Field *n* cannot reference field *m*, where *m* is greater than *n*, and field *m* does not have a `value` descriptor defined.

| | |
|---|---|
| `HAS_COLORS` | This variable evaluates to TRUE if **`fmli`** is invoked from a color terminal, otherwise it evaluates to FALSE. `HAS_COLORS` is placed in the UNIX shell environment. |
| `LININFO` | This variable evaluates to null if the current menu item or form field doesn't have a `lininfo` descriptor defined. Otherwise it evaluates to the value of the `lininfo` descriptor. (See Chapter 3, for an example of how to use this variable to output a help message for form fields or menu items.) |
| `LOADPFK` | When this variable is set to `yes`, `true`, or the null string, it directs FMLI to download alternative keystroke sequences into the function keys of a terminal that does not have fixed, preset values for them. `LOADPFK` is read from the UNIX shell environment. (See Appendix B for more information on automatic function key downloading.) |
| `MAILCHECK` | This variable determines the amount of time before a `SIGALRM` alarm automatically occurs. The minimum value for `MAILCHECK` is 120 seconds. If `MAILCHECK` is not defined, or defined as `0` (zero), it defaults to `300` seconds. `MAILCHECK` is read from the UNIX shell environment. |
| `NR` | This variable evaluates to the number of items in the menu frame. |
| `RET` | This variable evaluates to the exit value of the last executable run, whether in a backquoted expression or as the executable argument to the built-in utility **`run`**. If such an `exec` or `fork` system call fails, `RET` will be set to the sum of the return code of the `exec` or `fork` plus the integer 1000. |
| `SELECTED` | This variable evaluates to TRUE if the current item in a multi-select menu has been marked. It evaluates to FALSE if the item is not marked. |
| `TEXT` | This variable evaluates to the value of the `text` descriptor in a text frame. |

## Variable Evaluation

In frame definition files and application level files, variables are referenced by prefixing either `$` or `$!` to the variable name.

When you use the `$`*name* notation, the variable is evaluated only once. This implies that special characters lose their special meaning when they are coded in the values of strings. For example, if you assigned a value to the variable `VAR` as follows

```
`set -l VAR="`date` $HOME"`
```

and then requested FMLI to display the value of `$VAR`, the value displayed would be

```
`date` $HOME
```

When you use the `$!`*name* notation, the variable will be evaluated multiple times—as long as special characters remain in the expression. For example, if the variable `VAR` had the same value assigned as shown above, but you requested FMLI to display the value of `$!VAR`, the value displayed would be, for instance,

```
    Thu Sep 29 14:43:41 EDT 1989 /home/loginID
```

The `$!` notation should never be used when referencing the built-in variables (especially F1, F2, and so on), because it is impossible to guard against users entering special characters in form fields.

**NOTE**

Prior to FMLI Release 4.0, only the `$` notation existed for variable evaluation, and that notation exhibited the behavior now defined for `$!`.

For previously written FMLI applications now being run under FMLI Release 4.0 or later, a Boolean descriptor, `use_incorrect_pre4.0_behavior`, can be set in the initialization file if needed. This will cause FMLI to ignore the `$!` notation and interpret `$` in the old way. The default value (if not defined in the initialization file) for `use_incorrect_pre4.0_behavior` is FALSE.

This descriptor, and consequently the ability to make the `$` notation behave like the `$!` notation, will be removed in the next release of FMLI.

When a variable is evaluated that does not specifically reference a file, two environments are searched:

| | |
|---|---|
| local environment | This environment is specific to the current FMLI process (variables set with **`set -l`**). This is similar to an unexported shell variable. |
| UNIX system environment | The UNIX system environment is the standard UNIX environment. |

Whenever *environment* is referred to in this text, these environments are searched in the order listed.

Variable names must be referenced using one of the following formats:

| | |
|---|---|
| `$`*variable* or `${`*variable*`}` | Look for *variable* in the environment and evaluate to the value of that variable. |

**NOTE**

The built-in variable `F`*n* must be used with the format `{F`*n*`}` for fields greater than the ninth, that is, `{F10}`, `{F11}`, and so on.

| | |
|---|---|
| ${*variable*:-*default*} | Look for *variable* in the environment and if it is found evaluate to its value. If it is not found, evaluate to *default*. |
| ${(*filename*)*variable*} | Look for a line of the format *variable*=*value* in the file *filename*. If such a line is found, evaluate to *value*. |
| ${(*filename*)*variable*:-*default*} | Same as above, except if *variable* is not found anywhere, evaluate to *default*. |

Note that *filename* and *default* may themselves be variables, such as

```
${($HOME/.variables)NAME:-$LOGNAME}
```

## Descriptors

Descriptors are the basic building blocks of the Form and Menu Language. Each descriptor defines a particular attribute that you can customize for the type of frame you are defining. The three types of frames that you can define—menus, forms, and text frames—each have their own set of descriptors, as do the initialization file, the commands file, and the alias file.

The general syntax of descriptor statements in the Form and Menu Language is

*descriptor*=*value*

where *descriptor* is any valid descriptor for the frame definition file or application level definition file you are writing, and *value* is a value of the type expected by the descriptor. The value may include backquoted expressions that evaluate to part, or all, of the descriptor value, as well as FMLI commands and their arguments. Some descriptors have default values.

To take a simple example, if the variable `LOGNAME` evaluates to `chris`, then

```
name=hello there $LOGNAME --today is `date`
```

results in the value of the `name` descriptor being

```
hello there chris --today is Sun Aug 27 16:07:23 EDT 1989.
```

Note that there are no spaces around the equal sign (`=`) in a descriptor statement.

### Descriptor Evaluation

To obtain a descriptor value for the first time, FMLI must either use the default value of the descriptor, if any, or evaluate the expression coded for the descriptor. Expression evaluation resolves references to variables, removes quotes, and causes any backquoted expressions to be executed. Backquoted expressions may be used for their side effects only or may generate standard output that is used as part of the descriptor value.

By default, FMLI determines how often descriptors are evaluated. Most are evaluated only once, the first time the descriptor value is needed; the value of the descriptor remains the same throughout the life of the frame. This does not mean that the value cannot be used, or "referenced," again, only that it will not be recomputed each time it is referenced.

Other descriptors are evaluated multiple times in the life of a frame. In these cases, all backquoted expressions coded as part of the descriptor are executed, for output and side effects, each time the descriptor is evaluated. Some of these descriptors are evaluated whenever they are referenced, others are evaluated only when referenced in certain conditions. The `show` descriptor, for instance, is typically used to make fields in a form appear or disappear based on values the user has entered in other fields. `show` is referenced when the form is opened and thereafter each time the user navigates between fields in the form. `show` is evaluated, however, only the first time it is referenced and thereafter only when the user has changed the value of a field before navigating away.

For performance reasons, then, FMLI evaluates descriptors only as necessary in the typical case. That may still be too often, or not often enough, for your application. In these situations, you can use the `const` type cast to make sure that a descriptor is evaluated only once, no matter how many times it is referenced, or the `vary` type cast to make sure that a descriptor is evaluated whenever it is referenced.

As an example of how you might use `const`, consider a form that contains a field that should only be completed on Friday. You can define the `show` descriptor for the field so that it will appear only when the **`date +%a`** command evaluates to Friday:

```
show=`set -l DAY=date +%a; test "$DAY" = "Friday"`
```

In the example, the FMLI built-in utility **`set -l`** sets the local variable `DAY` to the output of the **`date +%a`** command.

The problem with this is that, by default, FMLI will evaluate `show` more times than is necessary for your application: not only when the form is opened, but whenever the user changes a value in a field and navigates to another field. To prevent that, you can use `const` as follows:

```
show=const `set -l DAY=date +%a; test "$DAY" = "Friday"`
```

`show` will be evaluated only when the form is opened.

`vary` is used to force a descriptor that is evaluated once by default, or only when it is referenced in certain conditions, to be re-evaluated each time it is referenced. Suppose you have defined a form that allows users to administer machines in a network. One field in the form displays a choices menu from which users can select the machine they want to act on. The field references a directory that contains files corresponding to each machine in the network. You can use the `rmenu` descriptor to define the choices for the field:

```
rmenu={ `ls $NetMachines` }
```

Suppose further, though, that machines will be added or removed from the directory list as necessary throughout the life of the form. That means your choices menu will have to change dynamically to reflect the changing contents of the directory. Because `rmenu` is evaluated only once by default, the code shown above will produce a choices menu that reflects the state of affairs when the form was opened, and not as it has changed since then. To produce a choices menu that changes dynamically, you use `vary` as follows:

```
rmenu=vary { `ls $NetMachines` }
```

`rmenu` will be evaluated whenever it is referenced.

## Descriptor Types

The types of descriptors are the following:

| | |
|---|---|
| Boolean | A descriptor of type Boolean must evaluate to either TRUE or FALSE. • FALSE is defined as the word “false,” irrespective of case, or a non-zero return code. • TRUE is defined as all values other than FALSE, as defined above. For example, `true`, `TRUE`, `yes`, `0`. |
| color | A descriptor of type color must evaluate to one of the following strings: `black`, `blue`, `green`, `cyan`, `red`, `magenta`, `yellow`, or `white`, or one you define using **`setcolor(1F).`** |
| command | A descriptor of type command must evaluate to an FMLI command, such as **open**, **nop**, **`exit`**. |
| integer | A descriptor of type integer must evaluate to an integer value. |
| layout | A descriptor of type layout (there is only one, `slk_layout`) must evaluate to one of only two values: either `3-2-3` or `4-4`. |
| null | A descriptor of type null exists only to get the side effect of a backquoted expression. Its value is ignored. |
| position | A descriptor of type position must evaluate to an integer value or one of the strings `any`, `center`, `current`, or `distinct`. |
| string | A descriptor of type string must evaluate to a sequence of characters. |

**NOTE**

If the integer value assigned to a descriptor that determines the offset of a frame or any of its components is greater than the boundaries of the screen work area, the frame will not be posted. The `begrow` and `begcol` descriptors are the exceptions to this. They default to `any`.

## Frame Definition File Descriptors

Menu, form, and text frame definition files have similar rules governing the order in which descriptors are defined.

| | |
|---|---|
| frame descriptors | Descriptors that apply to the frame as a whole must be defined first. There can be only one set of frame descriptors in a frame definition file. Each frame descriptor |

| | |
|---|---|
| | should be defined only once. If defined more than once, then the last instance of the descriptor in the set is used. Frame descriptors that are out of order, that is, any that follow item, field, or SLK descriptors in the frame definition file, are ignored. |
| item or field descriptors | Descriptors that apply to an item in a menu or a field in a form must be defined next. There can be multiple sets of item descriptors in a menu definition file, and multiple sets of field descriptors in a form definition file—as many sets as there are items or fields. In each set, however, each item or field descriptor should be defined only once. If defined more than once, then the last instance of the descriptor in the set is used.<br><br>Text frame definition files do not have an equivalent to item or field descriptors. |
| screen-labeled function key (SLK) descriptors | Descriptors that apply to screen-labeled function keys (the name that appears on the screen label as well as the function assigned to the function key) must be defined last. There can be multiple sets of SLK descriptors—as many sets as there are SLKs you are defining. In each set, however, a descriptor should be defined only once. If defined more than once, then the last instance of the descriptor in the set is used. |

The following tables summarize the available descriptors for each type of frame. They also show the default value of each descriptor, the type of string expected as a value, when the descriptor is referenced, and when it is evaluated by default. (See Chapter 3 and Chapter 4 for discussions of what these descriptors do and how to use them.)

**NOTE**

The “When Referenced” listing for each descriptor in the following tables should be considered a statement of the minimum number of times the descriptor is referenced. Many of the descriptors are referenced more times than is stated in the tables.

## Menu Descriptors

Table 2-1 lists the frame descriptors that can be used in a menu definition file. None of these descriptors is required in a menu definition file. If any are used they can be in any order, but they must precede the item descriptors.

**Table 2-1. Frame Descriptors for Menu Definition Files**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `altslks` | FALSE | Boolean | When menu is opened/updated | When menu is opened/ updated |
| `autosort` | FALSE | Boolean | When menu is opened/updated | When menu is opened/ updated |
| `begcol` | `any` | position | When menu is opened/updated | When menu is opened/ updated |
| `begrow` | `any` | position | When menu is opened/updated | When menu is opened/ updated |
| `close` | no default | null | When menu is closed for any reason | When menu is closed for any reason |
| `columns` | calculated value** | integer | When menu is opened/updated | When menu is opened/ updated |
| `done` | no default | command | When items are selected (not marked) in a multi-select menu; ignored in a single-select menu | Whenever referenced |
| `framemsg` | no default | string | When menu is opened/updated | When menu is opened/ updated |
| `help` | no default | command | When user asks for help | Whenever referenced |
| `init` | TRUE | Boolean | When menu is opened/updated | When menu is opened/ updated |
| `interrupt` | inherited value* | Boolean | When an interruptible descriptor is evaluated | Whenever referenced |
| `lifetime` | `longterm` | string | When menu is opened, closed, made current, or made non-current | Whenever referenced |
| `oninterrupt` | inherited value* | command | After descriptor evaluation is interrupted | Whenever referenced |

**Table 2-1. Frame Descriptors for Menu Definition Files (Cont.)**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `menu` | `Menu` | string | When menu is opened | When menu is opened |
| `multiselect` | FALSE | Boolean | When menu is opened/updated | When menu is opened/ updated |
| `reread` | FALSE | Boolean | When a **`checkworld`** occurs | Whenever referenced |
| `rows` | calculated value** | integer | When menu is opened/updated | When menu is opened/ updated |

Table 2-2 lists the item descriptors that can be used in a menu definition file. In each set of item descriptors, `name` is required and must be the first descriptor.

**Table 2-2. Item Descriptors for Menu Definition Files**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `action` | no default | command | When item is selected | Whenever referenced |
| `description` | no default | string | When menu is opened/updated | When menu is opened/ updated |
| `inactive` | FALSE | Boolean | When menu is opened/updated | When menu is opened/ updated |
| `interrupt` | inherited value* | Boolean | When `action` descriptor is evaluated | Whenever referenced |
| `itemmsg` | no default | string | When item is navigated to | Whenever referenced |
| `lininfo` | no default | string | When item is navigated to | Whenever referenced |
| `name` | no default | string | When menu is opened/updated | When menu is opened/ updated |

**Table 2-2. Item Descriptors for Menu Definition Files (Cont.)**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `oninterrupt` | inherited value* | command | After `action` descriptor evaluation is interrupted | Whenever referenced |
| `selected` | FALSE | Boolean | When menu is opened/updated | When menu is opened/updated |
| `show` | TRUE | Boolean | When menu is opened/updated | When menu is opened/updated |

Table 2-3 lists the descriptors that can be used to define screen-labeled function keys in a menu definition file. The `name` and `button` descriptors must be defined, and `name` must be first in each set of SLK descriptors.

**Table 2-3. SLK Descriptors for Menu Definition Files**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `action` | no default | command | When SLK is pressed | Whenever referenced |
| `button` | no default | integer | When menu is opened/updated | Whenever referenced |
| `interrupt` | inherited value* | Boolean | When SLK `action` descriptor is evaluated | Whenever referenced |
| `name` | no default | string | When menu is opened/updated | Whenever referenced |
| `oninterrupt` | inherited value* | command | After SLK `action` descriptor evaluation is interrupted | Whenever referenced |

* The value of `interrupt` and `oninterrupt` in any given set of descriptors is inherited from the next higher level in a precedence hierarchy. If these descriptors have not been defined anywhere in your application, `interrupt` defaults to FALSE and `oninterrupt` defaults to `` `message Operation interrupted!`nop ``. (See “Interrupt Signal Handling” on page 2-39 for more information.)

** The default value for `columns` and `rows` is determined by FMLI and depends in part on the number of items defined in the menu. (See Appendix A for a table describing the method of calculation.)

## Form Descriptors

Table 2-4 lists the frame descriptors that can be used in a form definition file. None of these descriptors is required in a form definition file. If any are used they can be in any order, but they must precede the field descriptors.

**Table 2-4. Frame Descriptors for Form Definition Files**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `altslks` | FALSE | Boolean | When form is opened/updated | When form is opened/ updated |
| `autolayout` | FALSE | Boolean | When form is opened/updated | When form is opened/ updated |
| `begcol` | `any` | position | When form is opened/updated | When form is opened/ updated |
| `begrow` | `any` | position | When form is opened/updated | When form is opened/ updated |
| `close` | no default | null | When form is closed | When form is closed |
| `done` | `close` | command | When form is saved | Whenever referenced |
| `form` | `Form` | string | When form is opened | When form is opened |
| `framemsg` | no default | string | When form is opened/updated | When form is opened/ updated |
| `help` | no default | command | When user asks for help | Whenever referenced |
| `init` | TRUE | Boolean | When form is opened/updated | Whenever referenced |
| `interrupt` | inherited value* | Boolean | When an interrupt-ible descriptor is evaluated | Whenever referenced |
| `lifetime` | `longterm` | string | When form is opened, closed, made current, made non-current | Whenever referenced |
| `oninterrupt` | inherited value* | command | After descriptor evaluation is inter-rupted | Whenever referenced |
| `reread` | FALSE | Boolean | When **`checkworld`** occurs | Whenever referenced |

Table 2-5 lists the field descriptors that can be used in a form definition file. In each set of field descriptors the `name` descriptor is required and must be first.

**Table 2-5. Field Descriptors for Form Definition Files**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `autoadvance` | FALSE | Boolean | When form is opened/updated | When form is opened/ updated |
| `choicemsg` | no default | string | When choices menu is selected | When choices menu is selected |
| `columns` | If `autolayout` is FALSE, -1. If `autolayout` is TRUE: 4 for first field, else previous field's value *** | integer | When form is opened/updated | When form is opened/ updated |
| `fieldmsg` | no default | string | When field is navi-gated to | Whenever referenced |
| `fcol` | If `autolayout` is FALSE, -1. If `autolayout` is TRUE: `1`+*current_ncol*+ *lengthOfLabel* if first field, or max of that and its value in previous field ** | integer | When form is opened/updated | When form is opened/ updated |
| `frow` | If `autolayout` is FALSE, -1. If `autolayout` is TRUE: *current_nrow* ** | integer | When form is opened/updated | When form is opened/ updated |
| `inactive` | FALSE | Boolean | When form is opened, made cur-rent, updated, saved | First time referenced and when referenced after an earlier field value has been changed |
| `invalidmsg` | Input is not valid | string | When `valid` eval-uates to false | First time referenced |
| `invalidOn-DoneMsg` | Relationship of val-ues in 2 or more fields is not valid | string | When `validOn-Done` evaluates to false | First time referenced |
| `lininfo` | no default | string | When this field is navigated to | Whenever referenced |
| `menuonly` | FALSE | Boolean | When form is opened/updated | When form is opened/ updated |

**Table 2-5.  Field Descriptors for Form Definition Files (Cont.)**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `name` | no default | string | When form is opened/updated | When form is opened/ updated |
| `ncol` | If `autolayout` is FALSE, -1. If `autolayout` is TRUE: 0 for first field, else previous field's value ** | integer | When form is opened/updated | When form is opened/ updated |
| `noecho` | FALSE | Boolean | When form is opened/updated | When form is opened/ updated |
| `nrow` | If `autolayout` is FALSE, -1. If `autolayout` is TRUE: 0 if first field of page or *previous_nrow*+ *previous_rows* ** | integer | When form is opened/updated | When form is opened/ updated |
| `page` | `1` *** | integer | When form is opened/updated | When form is opened/ updated |
| `rmenu` | no default | command | When form is opened/updated | When form is opened/ updated |
| `rows` | `1` *** | integer | When form is opened/updated | When form is opened/ updated |
| `scroll` | FALSE | Boolean | When form is opened/updated | When form is opened/ updated |
| `show` | TRUE | Boolean | When form is opened/updated and when any inter-field navigation occurs | First time referenced and when referenced after an earlier field value has been changed |
| `valid` | TRUE | Boolean | When interfield navigation is attempted from a changed field, or from any field with ENTER, and when form is saved | Whenever referenced |

**Table 2-5. Field Descriptors for Form Definition Files (Cont.)**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `validOnDone` | TRUE | Boolean | When form is saved | Whenever referenced |
| `value` | no default | string | When form is opened/updated | When form is opened/ updated |
| `wrap` | FALSE | Boolean | When form is opened/updated | When form is opened/ updated |

Table 2-6 lists the SLK descriptors that can be used in a form definition file. When they appear in a form definition file, they must be the last descriptors in the file. The `name` and `button` descriptors must be defined, and `name` must be the first descriptor in each set of SLK descriptors.

**Table 2-6. SLK Descriptors for Form Definition Files**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `action` | no default | command | When SLK is pressed | Whenever referenced |
| `button` | no default | integer | When form is opened/updated | Whenever referenced |
| `interrupt` | inherited value* | Boolean | When SLK `action` descriptor is evaluated | Whenever referenced |
| `name` | no default | string | When form is opened/updated | Whenever referenced |
| `oninterrupt` | inherited value* | command | After SLK `action` descriptor evaluation is interrupted | Whenever referenced |

* The value of `interrupt` and `oninterrupt` in any given set of descriptors is inherited from the next higher level in a precedence hierarchy. If these descriptors have not been defined anywhere in your application, `interrupt` defaults to FALSE and `oninterrupt` defaults to `` `message Operation interrupted!`nop ``. (See “Interrupt Signal Handling” on page 2-39 for more information.)

** A negative value for this descriptor will cause the label or input area being described to not appear in the form.

*** A zero or negative value for this descriptor will cause the field being described to not appear in the form.

## Text Frame Descriptors

Table 2-7 lists the frame descriptors that can be used in a text frame definition file. None of these descriptors is required in a text frame definition file. If any are used they can be in any order, but they must precede the SLK descriptors. Note that the only kinds of descriptors in text frame definition files are frame descriptors and SLK descriptors. (Text frames do not have an equivalent to item or field descriptors.)

Note that text frames may also be defined, at least for simple needs, using the **`textframe`** command. This command can be used to reduce the number of text frame definition files needed in an application. See “The textframe Command” on page 3-56 for more information on using this short-cut.

**Table 2-7. Frame Descriptors for Text Frame Definition Files**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `altslks` | FALSE | Boolean | When frame is opened/updated | When frame is opened/ updated |
| `begrow` | `any` | position | When frame is opened/updated | When frame is opened/ updated |
| `begcol` | `any` | position | When frame is opened/updated | When frame is opened/ updated |
| `close` | no default | null | When frame is closed | When frame is closed |
| `columns` | `30` | integer | When frame is opened/updated | When frame is opened/ updated |
| `done` | `close` | command | When frame is closed | Whenever referenced |
| `edit` | FALSE | Boolean | When frame is opened/updated | When frame is opened/ updated |
| `framemsg` | no default | string | When frame is opened/updated | When frame is opened/ updated |
| `header` | no default | string | When frame is opened/updated | When frame is opened/ updated |
| `help` | no default | command | When user asks for help | Whenever referenced |
| `init` | TRUE | Boolean | When frame is opened/updated | Whenever referenced |
| `interrupt` | inherited value* | Boolean | When an interruptible descriptor is evaluated | Whenever referenced |

**Table 2-7. Frame Descriptors for Text Frame Definition Files (Cont.)**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `lifetime` | `longterm` | string | When frame is opened, closed, made current, or made non-current | Whenever referenced |
| `oninterrupt` | inherited value* | command | After descriptor evaluation is interrupted | Whenever referenced |
| `reread` | FALSE | Boolean | When frame is opened/updated | Whenever referenced |
| `rows` | `min(10,`*linesOfText*) | integer | When frame is opened/updated | When frame is opened |
| `text` | no default | string | When frame is opened/updated | When frame is opened/updated |
| `title` | `Text` | string | When frame is opened | When frame is opened |
| `wrap` | TRUE | Boolean | When frame is opened/updated | When frame is opened/updated |

Table 2-8 lists the SLK descriptors that can be used in a text frame definition file. When they are used in a text frame definition file, they must be the last descriptors in the file. The `name` and `button` descriptors must be defined, and `name` must be the first descriptor in each set of SLK descriptors.

**Table 2-8. SLK Descriptors for Text Frame Definition Files**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `action` | no default | command | When SLK is pressed | Whenever referenced |
| `button` | no default | integer | When frame is opened/updated | Whenever referenced |
| `interrupt` | inherited value* | Boolean | When SLK `action` descriptor is evaluated | Whenever referenced |
| `name` | no default | string | When frame is opened/updated | Whenever referenced |
| `oninterrupt` | inherited value* | command | After SLK `action` descriptor evaluation is interrupted | Whenever referenced |

* The value of `interrupt` and `oninterrupt` in any given set of descriptors is inherited from the next higher level in a precedence hierarchy. If these descriptors have not been defined anywhere in your application, `interrupt` defaults to FALSE and `oninterrupt` defaults to `` `message operation interrupted!`nop ``. (See “Interrupt Signal Handling” on page 2-39 for more information.)

## Application Level File Descriptors

There are three kinds of application level files: the initialization file, the commands file, and the alias file. The rules which govern the order of descriptors are different for each and are covered in the appropriate section.

The following tables summarize the available descriptors for application level files. They also show the default value of each descriptor, the type of string expected as a value, when the descriptor is referenced, and when it is evaluated by default. (See Chapter 4 for discussions of what these descriptors do and how to use them.) Most descriptors for application level files are evaluated at initialization time. *Initialization time* can be either when the FMLI application session is started via the **`fmli`** command, or when the FMLI built-in utility **`reinit(1F)`** is executed. (The **`reinit`** utility parses and evaluates the descriptors in the file named as its argument and continues running the current application.)

### Initialization File Descriptors

The rules governing the order in which descriptors are defined in an initialization file are the following:

application descriptors
: Descriptors that apply to the application as a whole must be defined first. There can be only one set of application descriptors in an initialization file. In that set, each application descriptor should be defined once. If defined more than once, then the last instance of the descriptor in the set is used. Application descriptors fall into four functional groups:

- introductory frame descriptors
- banner line descriptors
- general application descriptors
- color descriptors

application SLK descriptors
: Descriptors that apply to application SLKs (the name that appears on the screen label as well as the function assigned to the function key) must be defined last in the initialization file. There can be multiple sets of application SLK descriptors—as many sets as there are SLKs you are defining. In each set, however, a descriptor should be defined only once.

#### Application Descriptors for the Initialization File

The following tables summarize the four functional groups of application descriptors.

Table 2-9 lists the descriptors that can be used to define an introductory frame in an initialization file.

**Table 2-9.  Introductory Frame Descriptors for the Initialization File**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `columns` | `50` | position | At initialization time | At initialization time |
| `rows` | `10` | position | At initialization time | At initialization time |
| `text` | no default | string | At initialization time | At initialization time |
| `title` | no default | string | At initialization time | At initialization time |

Table 2-10 lists the descriptors that can be used to define the banner line in an initialization file.

**Table 2-10.  Banner Line Descriptors for the Initialization File**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `bancol` | `center` | position | At initialization time | At initialization time |
| `banner` | no default | string | At initialization time | At initialization time |
| `working` | `Working` | string | At initialization time | At initialization time |

Table 2-11 lists the descriptors that can be used to define features of the application as a whole.

**Table 2-11. General Descriptors for the Initialization File**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `interrupt` | FALSE | Boolean | When an interruptible descriptor is evaluated | When an interruptible descriptor is evaluated |
| `nobang` | FALSE | Boolean | At initialization time | At initialization time |
| `oninterrupt` | `` `message Operation Interrupted!`nop `` | command | After descriptor evaluation is interrupted | After descriptor evaluation is interrupted |
| `permanentmsg` | no default | string | At initialization time | At initialization time |
| `slk_layout` | `3-2-3` | layout | At initialization time | At initialization time |
| `toggle` | `3` | integer | At initialization time | At initialization time |

**NOTE**

Prior to FMLI Release 4.0, only the `$` notation existed for variable evaluation, and that notation exhibited the behavior now defined for `$!`.

For previously written FMLI applications now being run under FMLI Release 4.0 or later, a Boolean descriptor, `use_incorrect_pre4.0_behavior`, can be set in the general descriptors section of an initialization file if needed. This will cause **`fmli`** to ignore the `$!` notation and interpret `$` in the way defined for `$!`. The default-if-not-defined value for `use_incorrect_pre4.0_behavior` is FALSE.

This descriptor, and consequently the ability to make the `$` notation behave like the `$!` notation, will be removed in the next release of FMLI.

Table 2-12 lists the descriptors that can be used to define the colors of various elements of the FMLI screen display in an initialization file.

**Table 2-12. Color Descriptors for the Initialization File**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `active_border` | `white` | color | At initialization time | At initialization time |
| `active_title_bar` | `black` | color | At initialization time | At initialization time |
| `active_title_text` | `white` | color | At initialization time | At initialization time |
| `banner_text` | `white` | color | At initialization time | At initialization time |
| `highlight_bar` | `black` | color | At initialization time | At initialization time |
| `highlight_bar_text` | `white` | color | At initialization time | At initialization time |
| `inactive_border` | `white` | color | At initialization time | At initialization time |
| `inactive_title_text` | `white` | color | At initialization time | At initialization time |
| `inactive_title_bar` | `black` | color | At initialization time | At initialization time |
| `screen` | `black` | color | At initialization time | At initialization time |
| `slk_text` | `white` | color | At initialization time | At initialization time |
| `slk_bar` | `black` | color | At initialization time | At initialization time |
| `window_text` | `white` | color | At initialization time | At initialization time |

## Application SLK Descriptors

Table 2-13 lists the descriptors used to define SLKs in the initialization file. When used in an initialization file, they must be the last descriptors in the file. The `name` and `button` descriptors must be defined, and `name` must be the first descriptor in each set of SLK descriptors.

**Table 2-13. Application SLK Descriptors for the Initialization File**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `action` | no default | command | When SLK is pressed | Whenever referenced |
| `button` | no default | integer | At initialization time | Whenever referenced |
| `interrupt` | inherited value* | Boolean | When a SLK `action` descriptor is evaluated | Whenever referenced |
| `name` | no default | string | At initialization time | Whenever referenced |
| `oninterrupt` | inherited value* | command | After SLK `action` descriptor evaluation is interrupted | Whenever referenced |

* The value of `interrupt` and `oninterrupt` in any given set of descriptors is inherited from the next higher level in a precedence hierarchy. If these descriptors have not been defined anywhere in your application, `interrupt` defaults to FALSE and `oninterrupt` defaults to `` `message Operation interrupted!`nop ``. (See “Interrupt Signal Handling” on page 2-39 for more information.)

## Commands File Descriptors

The commands file has only one set of descriptors, and that set can be defined multiple times. The maximum sets of descriptors that can be defined in a commands file is 64. The `name` descriptor must be first in each set and is required. Each descriptor should be defined only once in each set. If defined more than once, only the last instance is used.

Table 2-14 lists the descriptors that can be used in a commands file to redefine or disable FMLI commands, and define new commands.

**Table 2-14. Commands File Descriptors**

| Descriptor | Default if not Defined | Type | When Referenced | Default Frequency of Evaluation |
|---|---|---|---|---|
| `action` | no default* | command | At initialization time | Whenever referenced |
| `help` | no default | command | When user asks for help | Whenever referenced |
| `interrupt` | inherited value** | Boolean | When an `action` descriptor is evaluated | Whenever referenced |
| `name` | no default | string | At initialization time | Whenever referenced |
| `oninterrupt` | inherited value** | command | After `action` descriptor evaluation is interrupted | Whenever referenced |

* The `action` descriptor has no default unless `name` evaluates to a predefined command. In that case, it defaults to the predefined command.

** The value of `interrupt` and `oninterrupt` in any given set of descriptors is inherited from the next higher level in a precedence hierarchy. If these descriptors have not been defined anywhere in your application, `interrupt` defaults to FALSE and `oninterrupt` defaults to `` `message Operation interrupted!`nop ``. (See “Interrupt Signal Handling” on page 2-39 for more information.)

## FMLI Commands

An FMLI command is a command that is part of the Form and Menu Language and which forces a screen related operation to occur. FMLI commands cannot be executed in back-quoted expressions; however, a backquoted expression can generate an FMLI command. Only descriptors of type command can evaluate to an FMLI command: **`action`**, **`done`**, **`help`**, **`oninterrupt`**, and **`rmenu`**. A command descriptor must evaluate to a single FMLI command. If it does not, the terminal bell will sound. The FMLI commands that take arguments are noted (see “Syntax Notation” on page -iv for an explanation of syntax notation).

**NOTE**

The maximum number of arguments that may be given in an FMLI command is 25. Remember, however, that a frame can only reference the first 10 arguments (`ARG0-ARG9`).

## FMLI Commands: Syntax and Use

The following list briefly describes all the FMLI commands.

**`cancel`** [*frameID* . . .]

The **`cancel`** command evaluates the `close` descriptor of the specified frame and attempts to close the frame. When selected from the `Command Menu` it closes the previously current frame. **`cancel`** closes a frame without executing the `done` descriptor.

The *frameID* argument can be an integer identifying a frame or the path name of a frame definition file. If *frameID* is a path name, it can be relative or full, but it must match the path name used when the frame was opened. If *frameID* is not given, **`cancel`** closes the current frame.

**`checkworld`**

The **`checkworld`** command evaluates the `reread` descriptor for all open frames: any frame whose `reread` descriptor evaluates to TRUE is updated (see **`update`** command). This command is initiated by the `SIGALRM` signal every `MAILCHECK` seconds. It is also initiated by many other events, such as executing the **`open, close, goto, run,`** and **`unix-system`** commands, and frame-to-frame navigation. When the **`checkworld`** command is executed, the message line clears. (This side effect may confuse users, especially when they are not aware that a `SIGALRM` has occurred. A warning in your user documents may be warranted.)

**`choices`**

The **`choices`** command evaluates the `rmenu` and `choicemsg` descriptors (if defined) in the set of field descriptors defining the current field. If neither is defined, a message informs the user that no choices are available.

**`cleanup`**

The **`cleanup`** command evaluates the `lifetime` descriptor of all open frames and closes those for which `lifetime` evaluates to `shortterm` or `longterm`.

**`close`** [*frameID* . . .]

The **`close`** command evaluates the `lifetime`, `done`, and `close` descriptors of all frames named in the *frameID* argument list and closes them. Frames named as arguments when **`fmli`** is invoked, and those frames in which the `lifetime` descriptor evaluates to `immortal` (which means they close only when the user exits from the FMLI application) will remain open.

The **`close`** command has essentially the same functionality as the **`cancel`** command, and is a useful alternative when the **`cancel`** command has been disabled in the commands file. (Recall that disabling a command in the commands file makes it unavailable to developers as well as to users.)

The argument *frameID* must be an integer identifying the frame, or the path name of the frame to close. If *frameID* is a pathname, it can be relative or full, but it must match the path name used when the frame was opened. If *frameID* is not given, **`close`** removes the current frame.

**`cmd-menu`** The **`cmd-menu`** command opens the `Command Menu` frame, displaying it in the center of the work area.

**`done`** The **`done`** command evaluates the `done` descriptor (if it has been defined) in a frame. In menu, text and form frames, **`done`** is a descriptor of type command. If the `done` descriptor is not defined, it defaults to the **`close`** command.

**`exit`** The **`exit`** command evaluates the `close` descriptor for all open frames, and terminates the FMLI session.

**`frm-mgmt`** [*cmd* [*frameID*] ]

The **`frm-mgmt`** command allows you to move, reshape, or list currently open frames. It takes a maximum of two arguments, where *cmd* can be one of the sub-commands **`list`**, **`move`**, or **`reshape`**, and *frameID* is an integer or a path name identifying the frame (menu or text frames only) to act on if *cmd* is **`move`** or **`reshape`**. If *frameID* is a path name, it can be relative or full, but it must match the path name used when the frame was opened. If *frameID* is not given, a menu is displayed in the work area, from which a user can select **`list`**, **`move`**, or **`reshape`**. If the argument `list` is supplied, a frame will display a list of currently open frames. Selecting a frame from this list causes navigation to that frame. The argument `list` does not accept a *frameID* option.

The argument `move` allows a frame to be moved to a different location in the work area. The argument `reshape` will not work on a form frame, but menu frames or text frames can be reshaped and/or moved to a different location in the work area. If the *frameID* argument is not supplied to the sub-commands **`move`** and `reshape`, the operation occurs for the current frame when **`frm-mgmt`** is used on a descriptor line, or for the most recently current frame when a user selects **`frm-mgmt`** from the `Command Menu` or command line. If *frameID* is supplied, the operation occurs for the open frame with that *frameID.*

**`goto`** [*frameID*] The **`goto`** command makes another frame current. *frameID* is the number of a frame or the path name of the frame definition file. The path name can be relative or full, but it must match the path name used to open the frame. Users should only be told about the frame number argument.

The **`goto`** command is run when the command line is current and an integer is entered. For example, `CTRL-j 2` equates to **`goto 2.`**

**`help`**

The **`help`** command evaluates the `help` descriptor if one has been defined for the current frame. If one hasn't been defined the indicator flashes.

**`mark`**

The **`mark`** command marks or unmarks the current item in menus for which the `multiselect` descriptor evaluates to TRUE.

**`nextpage`**

The **`nextpage`** command pages forward one page in the current frame, if that frame understands paging, and if the user is not on the last page of the frame. If the user is on the last page of the frame the terminal bell sounds. In forms, a page comprises all fields defined to be on a given page of the form (via the `page` descriptor). In menus and text frames, a page is a frame full of information.

**`next-frm`**

The **`next-frm`** command makes the next frame the current frame. FMLI keeps a list of each frame that has been the current frame: the *next frame* in the list is the last frame opened from the current frame. Since the next frame is always relative to the current frame the order of the list does not always follow frame ID order.

**`nop`**

The **`nop`** command does nothing. Because descriptors of type command must eventually evaluate to an FMLI command, **`nop`** is useful in those cases where you want to specify a backquoted expression to evaluate, but you do not want to execute an FMLI command. The terminal will beep when a descriptor of type command does not evaluate to an FMLI command. Including **`nop`** in the descriptor definition will prevent the terminal from beeping, while invoking no other operation.

**`open`** [*type*] *filename* [*arg* . . .]

The **`open`** command opens a frame. The argument *type* can be one of the file type casts `MENU`, `FORM`, or `TEXT`, and indicates the type of frame to be opened. The argument *filename* is the path name of the frame definition file to be opened. The argument *arg* is a parameter that will be passed to the frame. In the following example

```
OPEN FORM $MYFRAMES/myform ARG1 ARG2
```

**`open`** opens a frame definition file **`$MYFRAMES/myform,`** identified as a form frame definition file by the file type cast `FORM`, and passes the parameters `ARG1` and `ARG2` to it. An example of passing parameters can be found in “Creating a Dynamic Menu” on page 3-23.

**`prev-frm`**

The **`prev-frm`** command makes the previous frame the current frame. FMLI keeps a list of each frame that has been the current frame: the *previous frame* in the list is the frame from which the current frame was opened. Since the previous frame is always relative to the current frame the order of the list does not always follow frame ID order.

**`prevpage`** The **`prevpage`** command pages backward one page in the current frame, if that frame understands paging, and if the user is not in the first page of the frame. In forms, a page comprises all fields defined to be on a given page of the form (via the `page` descriptor). In menus and text frames, a page is a screen full of information. If the user is in the first page of the frame the terminal bell sounds.

**`refresh`** The **`refresh`** command redraws the terminal screen. For example, **`refresh`** can be used if a broadcast message from the operating system corrupts the FMLI screen.

**`release`** The **`release`** command displays on the message line the release number of the version of FMLI you are currently running. The **`release`** command is meant to be used from the command line. Partial matching cannot be used with **`release`** (the command name must be typed in full).

**`reset`** The **`reset`** command causes the `value` descriptor of the current field to be re-evaluated, restoring the default value of the field if the current value is different. The descriptor is re-evaluated even if it has been modified by `const`.

**`textframe`** [*options*] The **`textframe`** command opens a simple text frame. It is a short-cut to using a full text frame definition file and can be coded in menu, form, and text frame definition files. The options correspond to the most commonly used text frame descriptors. The argument *text* is the text to be displayed in the text frame and may contain embedded newlines and tabs (including the `\n` and `\t` notations). See "The textframe Command" on page 3-56 for details on the options.

**`togslk`** The **`togslk`** command causes FMLI to display the set of SLKs that is not currently being displayed. It is a toggle between the two sets.

**`unix-system`** The **`unix-system`** command brings up the UNIX system shell in full screen mode.

**`update [`***frameID* [*mkcurr*]]

The **`update`** command forces a frame definition file to be reread regardless of the absence or value of the `reread` descriptor. If there are differences between what is read and what is on the screen, the frame will be redrawn. **`update`** will not reread the `menu`, `form`, or `title` descriptors. It takes two optional arguments, where *frameID* is an integer or a path name identifying the frame to update. If *frameID* is a path name, it can be relative or full, but it must match the path name used when the frame was opened. The argument *mkcurr* determines if the frame will be made current once the update is done. The argument *mkcurr* must be a Boolean value; if it is not given, it defaults to FALSE. If no arguments are given, **`update`** updates the current frame.

After **`update`** is executed in a menu frame, the cursor is positioned on the first menu item. In a form frame, the cursor is positioned on the first field of the first page of the form. In a text frame, the cursor is positioned on the first line of text.

## User Access to FMLI Commands

Users can execute some FMLI commands by selecting them from the `Command Menu`.

**Table 2-15. Default Assignments of FMLI Commands to the Command Menu**

| Command Menu | |
|---|---|
| cancel | next-frm |
| cleanup | prev-frm |
| exit | refresh |
| frm-mgmt | unix-system |
| goto | update |
| help | |

In addition, all FMLI commands can be executed from the command line. (Users can access the command line in an FMLI application by pressing `CTRL-j` or `CTRL-f c`.) FMLI commands that appear in the `Command Menu` or that are assigned to screen-labeled function keys should be explained in your user documentation. However, you should not document commands you do not want your users to use.

See Chapter 4 for information about how you can add commands to and disable commands in the `Command Menu` and how you can redefine the action assigned to a screen-labeled function key.

**NOTE**

When you disable an FMLI command in the commands file, the command becomes unavailable not only to users, but to developers. That is, you cannot use that command in frame definition files or application level files. In particular, do not disable the **`exit`** command.

Some FMLI commands also map, directly or indirectly, to default screen-labeled function keys.

**Figure 2-1. Default Assignments of FMLI Commands to Function Keys**

| Function Key | Menu Frame | Form Frame | Text Frame | Choices Menu | Command Menu |
|---|---|---|---|---|---|
| `F1` | help | help | help | | help |
| `F2` | mark* | choices | prevpage | | |
| `F3` | enter | save | nextpage | enter | |
| `F4` | prev-frm | prev-frm | prev-frm | | |
| `F5` | next-frm | next-frm | next-frm | | |
| `F6` | cancel | cancel | cancel | cancel | cancel |
| `F7` | cmd-menu | cmd-menu | cmd-menu | | |
| `F8` | chg-keys** | chg-keys** | chg-keys** | chg-keys** | chg-keys** |
| `F16` | chg-keys** | chg-keys** | chg-keys** | chg-keys** | chg-keys** |

* Function key `F2` is assigned the **`mark`** command only in multi-select menus. In single-select menus `F2` has no default assigned.

** Function keys `F8` and `F16` will default to **`chg-keys`** only if any of keys `F9` through `F15` are defined by the developer.

## Built-in Utilities

Built-in utilities provide often-needed programming functionality. By building them into FMLI they are more efficient to use than similar utilities provided in the UNIX system (that is, there is no need to fork a process to run them). FMLI recognizes built-in utilities in stand-alone backquoted expressions, and in backquoted expressions on descriptor lines.

FMLI built-in utilities return a Boolean value. It is FALSE if either the string `false` or a non-zero integer is returned, TRUE if 0 or any other string is returned. However, Boolean arguments to a utility follow standard format.

### Overview of the Built-in Utilities

Below is a brief summary of the FMLI built-in utilities. There are manual pages in section 1F for each.

**echo** The **echo** utility outputs its operands.

**fmlcut** The **fmlcut** utility is used to cut out selected fields of each line of a file. It has essentially the same functionality as the UNIX utility **cut**. It has been included as an FMLI built-in utility for performance reasons.

**fmlgrep** The **fmlgrep** utility is used to search for a certain pattern in a file. It has essentially the same functionality as the UNIX utility **grep**. It has been included as an FMLI built-in utility for performance reasons.

**fmlexpr** The **fmlexpr** utility evaluates its arguments as an expression, thus providing arithmetic and logical operations on integers, logical operations on strings, and some pattern matching facilities. It evaluates a single expression and writes the result to standard output. It has essentially the same functionality as the UNIX utility **expr**. It has been included as an FMLI built-in utility for performance reasons.

**fmlmax** The **fmlmax** utility is used to either determine the position of the field in a form or to determine the longest out of a number of strings. It is useful for the redesign of forms layout when the **autolayout** descriptor has been set to TRUE.

**getitems** The **getitems** utility takes as its only argument a delimiter string. It returns a list of the currently selected items, separated by the delimiter supplied.

**getfrm** The **getfrm** utility returns the current frame number. It takes no arguments.

**indicator** The **indicator** utility allows you to control the `Working` indicator and bell, and allows you to define your own indicators on the banner line.

**message** The **message** utility outputs its operands to the FMLI message line. The **-t** option outputs a *transient* message (lasts until another key is pressed), the **-f** option outputs a *frame permanent* message (lasts as long as the frame is current), and the **-p** option outputs a *permanent* message (lasts until another message is displayed, and reappears after that message clears). The terminal bell can also be made to sound.

**pathconv** The **pathconv** utility converts an alias to a full path name. It can also produce a shortened version of a path name suitable for use as a frame title.

**readfile**, **longline**

The **readfile** utility reads the file passed as its argument and writes it to standard output. If the system's locale is other than `C`, **readfile** tries to read <*dirname*>**/$LANG/**<*file*> if the `$LANG` subdirectory exists or <*dirname*>/<*file*> otherwise. After a call to **readfile**, a call to **longline** will return the length (including carriage return) of the longest line in the previously

| | |
|---|---|
| | read file. The **`longline`** utility can also take a file name argument, in which case it will return the length of the longest line in that file. |
| **`regex`** | The **`regex`** utility performs regular expression matching on its string input (utilizing **`regex`**. The **`regex`** utility is useful to dynamically generate the contents of a frame (see examples in Chapter 3). It can be used to approximate many of the capabilities of **`cut(1),`** **`paste(1),`** and **`grep(1),`** and some of the capabilities of **`sed(1).`** |
| **`reinit`** | The **`reinit`** utility takes as an argument the name of an initialization file. It is used to make global changes to the FMLI session while staying in the current application. |
| **`run`** | The **`run`** utility is used to invoke an executable in full-screen mode. |
| **`set, unset`** | These utilities set and unset variables either in the FMLI process, the UNIX system environment, or in files. |
| **`setcolor`** | The **`setcolor`** utility allows you to redefine an existing color, or define new colors if your terminal allows more than the eight colors already defined in FMLI. |
| **`shell`** | The **`shell`** utility is used to run a command using the UNIX system shell. Although it is not often needed in an FMLI application, it is useful when an application has an executable with the same name as an FMLI built-in or to run a UNIX system shell built-in. |
| **`test`** | The **`test`** utility checks to see if a condition is true. **`test`** is useful in conditional statements. It has essentially the same format as **`test`** in the UNIX system shell. |

Five other built-ins allow a frame or several frames (that is, form frames, menu frames, or text frames) to communicate to an external process through a pipe. The Form and Menu Language Interpreter will send strings to the external process and interpret the process's output accordingly. This capability is referred to as *co-processing*, and the built-in co-processing utilities are as follows:

| | |
|---|---|
| **`cocreate`** | The **`cocreate`** utility initializes communication to an independent co-process using named pipes. |
| **`cosend`** | The **`cosend`** utility sends strings from FMLI to the co-process. The **`-n`** option performs a *no wait* condition that sends text, but doesn't block for a response. |
| **`cocheck`** | The **`cocheck`** utility checks the incoming pipe for information. It returns TRUE or FALSE. |
| **`coreceive`** | The **`coreceive`** utility performs a “no wait” read on the pipe. It takes a process ID as an argument. |
| **`codestroy`** | The **`codestroy`** utility terminates the communication. |

For more information about how these co-processing utilities are used, see the **`coproc(1F)`** manual page.

## Conditional Statements

The Form and Menu Language provides a conditional statement for use within backquoted expressions that has the following syntax:

`if` *list* `then` *list* `[ elif` *list* `then` *list* `] . . . [ else` *list* `] fi`

where *list* is an optional newline, followed by a sequence of one or more FMLI statements, each of which must end with a semicolon.

Like conditional statements in the UNIX system shell language, the *list* following `if` is executed, and if the last command in the list has a zero exit status, then the *list* that follows `then` is executed. However, if the *list* following `if` has a non-zero exit status, the *list* following `else` will be executed. Multiple tests can be executed by using the `elif` clause. Conditional statements may be nested.

Output of a statement executed within an FMLI conditional construct can be redirected to a file specified after the statement.

The exit status of the conditional statement is the exit status of the last command executed in any `then` clause or `else` clause. If no such command was executed, the conditional statement returns a zero exit status.

**NOTE**

The FMLI conditional statement differs from the UNIX shell language conditional statement in some respects.

The UNIX shell language allows either a newline, a semicolon, or both, to end a statement, whereas FMLI only allows a semicolon.

Output of statements executed within an FMLI conditional construct cannot be piped to a statement following the `fi` of the construct. Similarly, output redirection to a file specified after the `fi` of the construct does not work.

An FMLI conditional statement cannot occur following the && or `||` operators in a given backquoted expression.

(See “Built-in Utilities” on page 2-34 and the manual pages in section 1F for information on **`test`** and **`fmlexpr`**.)

Conditional statements are useful when only one of several actions is appropriate, based on user input. For example, assume that you have defined a form in which a user must enter either `1` or `2` in the first field to display one of two appropriate text frames. A conditional statement can be used to display an appropriate error message via the use of the descriptor `invalidmsg`, when the user enters an invalid response:

```
invalidmsg=`if [ $F1 -gt 2 ];
  then
    echo “APPLID:nnn: Selection code has to be less than 3”;
  else
    echo “APPLID:nnn: Selection code has to be more than 0”;
  fi`
```

where `$F1` (an FMLI built-in variable) evaluates to the current value of the first field, **`APPLID`** is the name of your application, and `nnn` is the message sequence number.

Another conditional statement can be used to open the text frame requested by the user. The descriptor `done` in the same form definition file would be defined as follows:

```
    done=`if [ $F1 = 1 ];
          then echo “open text dir.explain”;
          else echo “open text file.explain”;
          fi`
```

## Signal Handling

The following signals are caught by FMLI:

| | | |
|---|---|---|
| SIGABRT * | SIGIOT * | SIGTSTP |
| SIGALRM | SIGINT * | SIGTTIN |
| SIGBUS | SIGPIPE * | SIGTTOU |
| SIGEMT | SIGQUIT * | SIGUSR1 |
| SIGFPE | SIGSEGV | SIGUSR2 |
| SIGHUP * | SIGSYS | SIGXCPU |
| SIGILL | SIGTERM * | SIGXFSZ |

Those signals marked by an asterisk (`*`) are caught only if the application that invokes FMLI has not caused them to be ignored. If an FMLI application is terminated by a signal, the terminal **`stty(1)`** settings will be reset and the message “*progname* `terminated by signal:` *description*” will be printed, where *progname* is the basename of the executable invoked to run the application and *description* is the string provided by **`psignal(3C).`** Not all of the signals listed above cause an application to terminate. These normally won't—`SIGALRM, SIGINT, SIGTSTP, SIGTTIN, SIGTTOU, SIGUSR1,` and `SIGUSR2`—although they can indirectly cause an FMLI application to terminate. For example, `SIGUSR2`, which is generated by the **`vsig`** command, could cause the application to execute the **`exit`** command. See the **`intro(2)`** manual page for a complete discussion of signals.

## Interrupt Signal Handling

Prior to Release 4.0 of FMLI, the only way an executable invoked from within an FMLI application could be interrupted was if it was invoked as an argument to the built-in utility **`run`** (except when the **`-s`** option is specified).

In FMLI Release 4.0 or later, executables initiated in a backquoted expression that is the value of either the `action` or `done` descriptors—wherever they occur in your application—can also be designated as interruptible through the use of the `interrupt` descriptor. No other descriptors of type command (`help`, `rmenu`, and `oninterrupt`) are affected by the `interrupt` descriptor.

The Boolean descriptor `interrupt` defines whether or not an executable can be interrupted by the user. It always defaults to FALSE. A companion to `interrupt`, the command descriptor `oninterrupt`, defines what will be done when the interrupt signal (`SIGINT`) is received. It always defaults to `` `message Operation interrupted!`nop.`` The value of `oninterrupt` can be any action normally permitted for a command descriptor (both backquoted expressions and FMLI commands). The `oninterrupt` descriptor is ignored if `interrupt` has not been defined anywhere in your application or if `interrupt` evaluates to FALSE.

Depending on the kind of frame definition file or application level file they are used in, these two descriptors are independently subject to one of several inheritance hierarchies (see Figure 2-16, a table of inheritance hierarchies for `interrupt` and `oninterrupt`).

**NOTE**

The status of the `interrupt` descriptor only affects executables in backquoted expressions. Its status does not affect FMLI commands (such as **`cancel`**), built-in utilities other than **`run`** (such as **`shell`**), or any of their child processes. Nor does it affect processes run from FMLI that “take over the screen,” such as the shell obtained by using the **`unix-system`** command from the command menu. (In these cases, interrupt handling is done by the full-screen application.)

The interrupt status in effect for an `action` or `done` descriptor applies to all executables in the descriptor. However, if the executable that is being executed when the interrupt signal is received has itself been coded to ignore interrupts, it will complete its normal execution, but remaining commands in the descriptor will not be executed. (FMLI built-in utilities behave the same as executable utilities that have been coded to ignore interrupts.) When an executable is interrupted, any commands (built-in utilities and FMLI commands, as well as other executables) remaining in the descriptor are ignored by the interpreter. Instead, whatever you have defined to be the value of the `oninterrupt` descriptor will be executed.

The scope of `interrupt` is independent from the scope of `oninterrupt`, and each depends on where it is coded. The highest level in the inheritance hierarchy is the one in effect for the current `action` or `done` descriptor. For example, coding `interrupt=true` once, in the general descriptors section of an initialization file, means that all executables in any `action` or `done` descriptor anywhere in your application (including any in the commands file or defined for SLKs) can be interrupted by the user.

Continuing this example, if `interrupt=false` is then coded with the frame descriptors in a menu definition file, then the status of FALSE is inherited by all items defined for that menu, while all executables in all other frames remain interruptible. Going one step further, if `interrupt=true` is coded with the item descriptors for one item in that menu, then executables coded in that item's `action` descriptor will be interruptible by users, and all other items will remain uninterruptible. Inheritance of the value of the `oninterrupt` descriptor is handled the same way, but is completely distinct from the `interrupt` descriptor.

These descriptors can also be defined in the commands file (see Chapter 4 for a discussion of the commands file). If `interrupt` or `oninterrupt` are not defined for a command in the commands file, that command will inherit the value of `interrupt` and/or `oninterrupt` defined with the general descriptors in the initialization file. If not defined there, the FMLI default value is inherited.

Inheritance of these two descriptors is handled slightly differently for screen-labeled function keys. Redefining a SLK in a frame definition file completely overrides a definition of it you may have coded in the initialization file. For example, if you define `interrupt` for a particular SLK in the initialization file, but do not include `interrupt` in a redefinition of that SLK in a frame definition file, the SLK will inherit the value of the `interrupt` descriptor defined at the next lower inheritance level (from the frame descriptors if defined there, then from the general descriptors in the initialization file if defined there, then from the FMLI defaults).

The table in Table 2-16 summarizes the inheritance hierarchies for both descriptors wherever they can be used:

**Table 2-16. Inheritance Hierarchies Used to Determine the Values of interrupt and oninterrupt When Interrupt Key Is Pressed**

| Inheritance Level | Executable Code | | | |
|---|---|---|---|---|
| | `action` Descriptor in: | | | `done` Descriptor in: |
| | menu items | SLK definitions | Command definitions | any frame |
| 1 (lowest) | FMLI defaults | FMLI defaults | FMLI defaults | FMLI defaults |
| 2 | values coded with the general descriptors in an initialization file | values coded with the general descriptors in an initialization file | values coded with the general descriptors in an initialization file | values coded with the general descriptors in an initialization file |
| 3 | values coded with the frame descriptors in a menu definition file | values coded with the frame descriptors in a frame definition file | values coded in the commands file | values coded with the frame descriptors in a frame definition file |
| 4 | values coded with the item descriptors in a menu definition file | values coded with the SLK descriptors in an initialization file* | n/a | n/a |
| 5 (highest) | n/a | values coded with the SLK descriptors in a frame definition file* | n/a | n/a |

* SLK-specific descriptors in a frame definition file completely override SLK descriptors defined in the initialization file. If either the `interrupt` or `oninterrupt` descriptor, or any other SLK-specific descriptor, is coded at level 5 (highest inheritance level), then all SLK-specific descriptors coded at level 4 are ignored.

## Terminal Display Attributes

The terminal display attributes and the alternate character set defined in **`curses(3curses)`** are supported in FMLI. If the terminal your application is being run on does not have these capabilities then they are approximated as best as possible by **`curses`**. If the terminal is capable of outputting graphic characters, inverse video, bold/dim, and so on, then these attributes will be available in FMLI, in the following places:

- in text frame definition files, in text defined in the `text` and `header` descriptors

- in form definition files, in the value assigned to the field-level descriptor `name`
- in the initialization file, in the value assigned to the `banner` descriptor
- in arguments to the **`message`** built-in utility
- in the argument to the **`indicator`** built-in utility

The character sequence to turn a terminal attribute on in FMLI applications is of the form `\+`*xx*. To turn the attribute off, the form `\-`*xx* is used. If *xx* is not a valid FMLI character sequence for a terminal attribute, the entire character sequence, as coded, is output.

Table 2-17 lists the FMLI character sequence names and maps them to the applicable **`curses`** defined constant.

**Table 2-17. Table of FMLI Character Sequences for Display Attributes**

| FMLI Character Sequence | **`curses`** defined-constant | Description |
|---|---|---|
| `so` | `A_STANDOUT` | terminal's best highlighting mode |
| `ul` | `A_UNDERLINE` | underlining |
| `rv` | `A_REVERSE` | reverse video |
| `bk` | `A_BLINK` | blinking |
| `dm` | `A_DIM` | half-bright |
| `bd` | `A_BOLD` | extra bright or bold |
| `ac` | `A_ALTCHARSET` | alternate character set |
| `nm` | `A_NORMAL` | reset all attributes to off |

## Using the Alternate Character Set

If the alternate character set attribute has been turned on in the text to be displayed (using `\+ac),` the alternate character set for line drawing (glyphs) will be displayed. This character set is shown in Table 2-18.

**Table 2-18. Alternate Character Set**

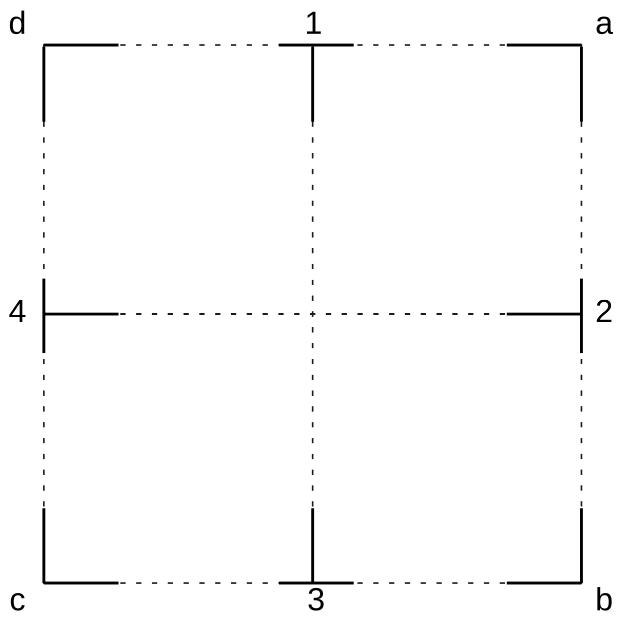
| Character | Default* | Glyph Description |
|---|---|---|
| a | + | upper right corner** |
| b | + | lower right corner** |
| c | + | lower left corner** |
| d | + | upper left corner** |

**Table 2-18. Alternate Character Set (Cont.)**

| Character | Default* | Glyph Description |
|---|---|---|
| 1 | + | top tee** |
| 2 | + | right tee** |
| 3 | + | bottom tee** |
| 4 | + | left tee** |
| - | - | horizontal line** |
| \| | \| | vertical line** |
| + | + | plus** |
| < | < | arrow pointing left** |
| > | > | arrow pointing right** |
| v | v | arrow pointing down** |
| ^ | ^ | arrow pointing up** |
| # | : | checkerboard (stipple)** |
| O | # | solid square block† |
| I | # | lantern symbol† |
| ' | + | diamond† |
| f | ' | degree symbol† |
| g | # | plus/minus† |
| h | # | board of squares† |
| o | - | scan line 1† |
| s | _ | scan line 9† |
| ~ | o | bullet† |

* The defaults listed in this column are the ASCII characters that will be displayed if the terminal does not support an alternate character set, or if that particular character is not implemented in that set.

** The character used to obtain this glyph is different in FMLI from the default character used in **`terminfo(4)`** because we feel these are easier to remember. The following diagram illustrates the first eight glyphs:



† This glyph is not supported by all terminals.

An example of the use of terminal display attributes is given in “Example Text Frame Definition Files” on page 3-57.

# 3
# Frame Definition Files

# 3
# Frame Definition Files

## Introduction

This chapter explains each of the descriptors that you can define for menu frames, form frames, and text frames.

- “Menu Frame Descriptors” on page 3-1 describes the functionality of the frame and item descriptors for menus.
- “Examples of Menu Definition Files” on page 3-8 presents some of the different ways you can create menu contents, and customize their appearance.
- “Form Frame Descriptors” on page 3-26 covers the frame and field descriptors for forms.
- “Example Form Definition Files” on page 3-42 gives examples of their use.
- “Text Frames” on page 3-51 describes the frame descriptors for text frames (text frames have only frame descriptors and SLK descriptors).
- “Example Text Frame Definition Files” on page 3-57 gives examples of their use.
- “Other Useful Examples” on page 3-61 covers some aspects of frame definition files that can be equally useful in all three types of frame definition files.

**NOTE**

Although SLK descriptors can be used in menu, form, and text frame definition files, they are discussed in Chapter 4. Their use as described there applies to frame definition files as well.

## Menu Frame Descriptors

A menu frame definition file can begin with an optional set of frame descriptors (one set per menu), followed by at least one set of item descriptors (one set per item in the menu), and it can end with one or more optional sets of SLK descriptors defining SLKs to be displayed when the menu is current (one set per screen-labeled function key).

Some of the attributes of a menu that you can define are the following:

- the action to take when the menu is opened
- whether the user may select more than one item (multi-select)
- whether to open a multi-select menu with specific items already selected (marked)
- where to place the menu on the screen
- the longevity of the menu
- whether or not to show a specific item
- the action to take for each item
- the action to take when the menu is closed

The descriptors in a menu definition file must follow this order:

```
[frame_descriptor_1
.
.
.
frame_descriptor_n]

item-one_descriptor_1
.
.
.
item-one_descriptor_n

[item-n_descriptor_1
.
.
.
item-n_descriptor_n]

. . .

[SLK-n_descriptor_1
.
.
.
SLK-n_descriptor_n

. . .]
```

**NOTE**

Out-of-order descriptors will be ignored if this order—frame, then items, then SLKs—is not followed.

## Frame Descriptors for Menus

The optional set of frame descriptors can include any valid frame descriptor, in any order. Each of these descriptors should be used only once in a menu definition file. If defined more than once in the set, the last one is used. In the following explanations, FALSE is defined as the word “false,” irrespective of case, or a non-zero return code. The notation TRUE is defined as all values other than FALSE as defined above (for example, `true`, `TRUE`, `yes`, `0`).

| | |
|---|---|
| `altslks` | The `altslks` descriptor defines whether SLKs 9 through 16 are displayed when the frame is initially opened. If `altslks` evaluates to TRUE, SLKs 9 through 16 will be displayed. The default, if this descriptor is not defined, is FALSE, which causes SLKs 1 through 8 to be displayed. |
| `autosort` | The boolean descriptor `autosort` defines whether the items in a menu should be sorted. This might be sensible in international applications when menu items should appear in alphabetical order irrespective of the current locale. |
| `begrow`, `begcol` | The `begrow` and `begcol` descriptors define the original position of the top left corner of the menu frame in the user's work area. (`begrow=0` and `begcol=0` evaluates to the upper left corner of the work area.) These descriptors accept values of type position: |

| | |
|---|---|
| `center` | the menu frame will be centered in the work area |
| `current` | the menu frame overlaps the current frame's position (valid for `begrow` only) |
| `distinct` | the menu frame will not overlap the current frame if possible (valid for `begrow` only) |
| `any` | FMLI chooses a position with least amount of total overlap |
| *integer* | the menu frame will be positioned in an absolute position, defined by *integer*. Defining `begrow` and `begcol` to be integer values causes the frame to appear in the given position. |

If either `begrow` or `begcol` evaluates to `center`, then the other can only be an integer value or `center`. Any other value is ignored and the descriptor defaults to `center`.

If neither is `center`, then the value of `begrow` determines the value of `begcol`; if `begrow` is `current`, `distinct`, `any`, or an invalid value, then `begcol` defaults to `any`. If `begrow` is a valid integer, `begcol` can be a valid integer; if `begcol` is an invalid integer in this case, it defaults to `any`. If integer values are supplied and either `begrow` or `begcol` are outside the screen boundary, a default value of `any` will be used.

`close`

The `close` descriptor is evaluated when the menu is closed and/or when the user exits from the FMLI application. The `close` descriptor is of type null, which means its only purpose is to obtain the side effects of backquoted expressions coded in its definition.

`columns`

The `columns` descriptor defines the number of items displayed in a row of a menu frame. It must evaluate to a positive integer; if it doesn't, `columns` will be ignored. It will also be ignored if `description` is defined for any menu item. If neither `columns` nor `rows` is defined, menu dimensions will be determined by the interpreter. Given `columns`, the number of rows needed to display the items in the menu is calculated. If there is a conflict between the value provided by the `rows` descriptor and the calculated value, the calculated value takes precedence. Menu item names will be truncated, if required, to fit in the specified `columns`. (See Appendix A for a complete discussion of the method used by FMLI to calculate rows and columns.)

**NOTE**

This descriptor should not be specified for dynamically generated menus if there is no way to guarantee that menu items will not be truncated.

`done`

The `done` descriptor defines the action to be executed when the user presses `ENTER` in a multi-select menu. This descriptor is ignored in a single-select menu.

`framemsg`

The `framemsg` descriptor displays its value on the message line as long as the menu is current. It can be temporarily replaced by

- a message displayed when the `itemmsg` descriptor is defined for a specific item in the menu frame
- a message generated by the **`message`** built-in utility with the **`-t`** (default) option
- an FMLI error message

It can be replaced for as long as the frame is current by a message generated by the **`message`** built-in utility with the **`-f`** option. (See the **`message(1F)`** manual page.)

`help`

The `help` descriptor specifies what will happen when a user asks for help while in the menu. Since this descriptor is evaluated when the user requests help, the specification of what help is displayed can be determined through parameters that are set interactively.

`init`

The `init` descriptor defines whether the menu frame will be opened. If this descriptor is not defined, it defaults to TRUE, which means the menu frame will be opened. If `init` evaluates to FALSE, the menu frame will not be opened, but the `close` descriptor will be evaluated. If `init` evaluates to FALSE on an update, the frame is closed, unless it is an initial frame.

`interrupt`

The Boolean descriptor `interrupt` defines whether an executable that is coded in `action` or `done` descriptors can be interrupted by users (FALSE means not interruptible, TRUE means interruptible). It is subject to an inheritance hierarchy: if not defined anywhere in your application, the default value FALSE applies throughout. If explicitly defined at any inheritance level, then executables in `action` and `done` descriptors at or above that inheritance level will inherit that defined value. (See “Interrupt Signal Handling” on page 2-39 for complete information.)

If defined among the frame descriptors in a menu definition file, that value of `interrupt` is inherited by all sets of item descriptors and all sets of SLK descriptors in the menu, unless it is redefined for a specific item or SLK.

`lifetime`

The `lifetime` descriptor defines when the menu frame will be closed (that is, removed from the work area). It is evaluated whenever the menu is opened, closed, made current, or made non-current. The acceptable values are:

| | |
|---|---|
| `shortterm` | the menu closes whenever the user navigates to another frame or when the command line is accessed (the user presses `CTRL-j` or `CTRL-f c`) |
| `longterm` | the menu closes when the user issues a **`cleanup`** or **`close`** command |
| `permanent` | the menu closes whenever the user issues a **`close`** command |
| `immortal` | the menu closes only when the user exits from the application |

The `lifetime` descriptor is ignored in menu definition files given as arguments when **`fmli`** is invoked. Such menus have a lifetime of `immortal`. See “Other Useful Examples” on page 3-61 for an example of how this descriptor may be used to close a frame when another is updated.

`menu`

The `menu` descriptor defines the title of the menu that appears in the frame's title bar. If not defined in the frame definition file, it defaults to `Menu`. It will be truncated if it is longer than `DISPLAYW-6`.

`multiselect`

The `multiselect` descriptor defines whether the menu is a multi-select menu. A multi-select menu allows the user to select more than one menu item. When this descriptor evaluates to TRUE, the SLK `F2` will map to the **`mark`** command, and the nature of the `action` descriptor changes for all items (see the description of `action` in “Item Descriptors for Menus” on page 3-6).

`oninterrupt`

The command descriptor `oninterrupt` defines what will happen when an interrupt signal is received. If `interrupt` is not coded anywhere in your application, or if it evaluates to FALSE, `oninterrupt` is ignored.

`oninterrupt` is subject to an inheritance hierarchy: if not defined anywhere in your application, the default value `` `message Operation interrupted!` nop `` applies throughout. If explicitly defined at any inheritance level, then executables in `action` and `done` descriptors at or above that inheritance level will inherit that defined value. (See “Interrupt Signal Handling” on page 2-39 for complete information.)

For example, if defined among the frame descriptors in a menu definition file, the value of `oninterrupt` is inherited by all sets of item descriptors and all sets of SLK descriptors in the menu, unless it is redefined for a specific item or SLK.

`reread`

If `reread` is not defined, it defaults to FALSE. If `reread` evaluates to TRUE, the menu will be periodically updated by rereading its description file when the **`checkworld`** command is executed. **`checkworld`** is executed when a `SIGALRM` alarm occurs (every `$MAILCHECK` seconds). Other times **`checkworld`** is executed include when a frame is opened, closed, or navigated to. (See **`checkworld`** in “Built-in Variables” on page 2-7.) When **`checkworld`** occurs, all frames whose `reread` descriptor evaluates to TRUE will be updated. (However, the `menu` descriptor is not reread.) Execution of **`checkworld`** may cause the message line to clear.

`rows`

The `rows` descriptor defines the desired number of rows long a menu frame will be. It must evaluate to an integer value greater than 0 and less than `DISPLAYH-2`; if it doesn't, `rows` will be ignored. If neither this descriptor nor `columns` is defined, menu dimensions will be determined by FMLI. Given `columns`, the number of rows needed to display the items in the menu is calculated. If there is a conflict between the value provided by the `rows` descriptor and the calculated value, the calculated value takes precedence. (A table summarizing these calculations can be found in Appendix A.)

## Item Descriptors for Menus

In each set of item descriptors, the `name` descriptor must be first; but others may be in any order. If a descriptor appears more than once in a set, the last one is used.

`action`

The `action` descriptor defines an action to be executed when the user selects this item in a single-select menu. Multiple backquoted expressions are allowed, as they are with any descriptor, but the final value of this descriptor must be a single FMLI command.

If the menu is multi-select (`multiselect=true`), the nature of this descriptor changes: FMLI commands are ignored if defined in this descriptor; however, backquoted expressions are executed when the item is marked.

`description`

The `description` descriptor defines a string which is displayed to the right of the item name but which is not highlighted when the cursor is on the item. When this descriptor is defined for any item in a menu, that menu will automatically display a single column of items, even if `columns` is defined.

`inactive`

The `inactive` descriptor defines an item as inactive when the menu is displayed. An item that is inactive cannot be navigated to, and consequently cannot be selected or un-selected. If not defined, `inactive` defaults to FALSE. If this descriptor evaluates to TRUE, the item is displayed with half-bright attribute (on most terminals). In multi-select menus, an inactive item can be selected only if the selected descriptor evaluates to TRUE for the item.

`interrupt`

The Boolean descriptor `interrupt` defines whether an executable that is coded in `action` or `done` descriptors can be interrupted by users (FALSE means not interruptible, TRUE means interruptible). It is subject to an inheritance hierarchy: if not defined anywhere in your application, the default value FALSE applies throughout. If explicitly defined at any inheritance level, then executables in `action` and `done` descriptors at or above that inheritance level will inherit that defined value. (See “Interrupt Signal Handling” on page 2-39 for complete information.)

If defined in a set of item descriptors in a menu definition file, that value of `interrupt` is inherited only by that menu item.

`itemmsg`

The `itemmsg` descriptor defines information that will be displayed on the message line when the item is navigated to. The `itemmsg` descriptor displays a message with transient duration. That is, it remains on the message line only until the user presses another key or a **`checkworld`** occurs. Transient messages take precedence over frame duration messages and permanent duration messages (see the **`message(1F)`** manual page for more information).

`lininfo`

The `lininfo` descriptor defines a string that will be assigned to the local environment variable `LININFO` when the user selects this menu item. If `lininfo` is not defined, `LININFO` evaluates to null. In multi-select menus, when the `getitems` built-in utility is executed, if `lininfo` is defined and this item is marked, its value is output.

`name`

The `name` descriptor defines a string that will appear in the menu, identifying the menu item. This string is highlighted when the item is navigated to. For multi-select menus, when the `getitems` built-in utility is executed, and the `lininfo` descriptor has not been defined for this marked item, the value of the `name` descriptor is output.

`oninterrupt` The command descriptor `oninterrupt` defines what will happen when an interrupt signal is received. If `interrupt` is not coded anywhere in your application, or if it evaluates to FALSE, `oninterrupt` is ignored.

`oninterrupt` is subject to an inheritance hierarchy: if not defined anywhere in your application, the default value `` `message Operation interrupted!` nop `` applies throughout. If explicitly defined at any inheritance level, then executables in `action` and `done` descriptors at or above that inheritance level will inherit that defined value. (See “Interrupt Signal Handling” on page 2-39 for complete information.)

If defined in a set of item descriptors in a menu definition file, that value of `oninterrupt` is inherited only by that menu item.

`selected` The `selected` descriptor defines whether a menu item in a multi-select menu should default to selected (TRUE) or not selected (FALSE) when the menu is opened. If `selected` evaluates to TRUE, the item is marked with the selected icon (an asterisk) when the menu is opened. If this descriptor is not defined, it defaults to FALSE. This descriptor is ignored when `multiselect` evaluates to FALSE (that is, in single-select menus).

`show` The `show` descriptor defines whether this menu item will be displayed. If this descriptor is not defined, it defaults to TRUE, and the menu item will be displayed. If it evaluates to FALSE, the menu item will not be displayed.

**NOTE**

Screen labels and actions for function keys can be defined in a menu description file as well as in the initialization file. Each set of screen-labeled function key descriptors must include the `name`, `button`, and `action` descriptors, and `name` must be first. If a descriptor appears more than once in a set, the last one is used.

See Chapter 4 for a discussion of how to use the screen-labeled function key descriptors.

## Examples of Menu Definition Files

The following examples show you how to write menu definition files. Refer to “Writing an Internationalized Application” on page 1-12 when writing internationalized applications.

## Defining a Simple Menu

A menu definition file usually starts with a set of descriptors that pertain to the entire menu, known as frame descriptors. Frame descriptors are optional in menu definition files. If you choose not to define any explicitly, their default values apply to the menu.

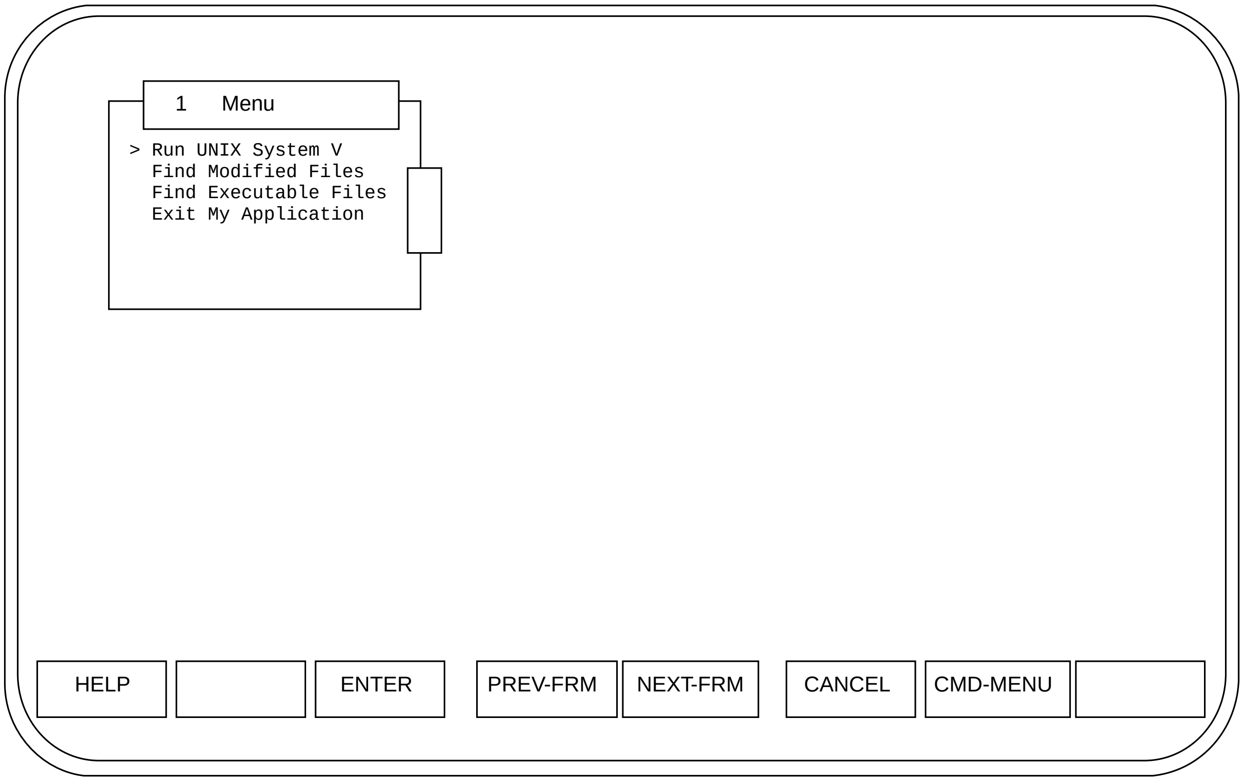
The following menu definition file has no frame descriptors, and defines four simple menu items. (Blank lines between logical sections of frame definition files are recommended for readability.)

```
name="Run UNIX System V"
action=unix-system

name="Find Modified Files"
action=`find $HOME -mtime -7 -print > modfiles`nop

name="Find Executable Files"
action=`find $HOME -perm -100 -print > execfiles`nop

name="Exit My Application"
action=exit
```

**Figure 3-1. Menu.items: An Example of Menu Item Descriptors**

The first item definition permits the user to access a full-screen UNIX system shell. The second item definition runs **`find(1)`** to locate files whose contents have been modified within the past seven days and saves the list of filen ames in a file named **`modfiles.`** The third item runs **`find`** to locate files that are executable by the owner and also saves the output in a file named **`execfiles`**. The fourth menu item permits the user to exit from the application.

Because no frame or SLK descriptors were defined, their default values apply: for example, `menu` is a frame descriptor used to define a title for a menu. Since `Menu.items` did not explicitly define it, the default value `Menu` is displayed in the title bar. This menu would appear as follows:



```
> Run UNIX System V
  Find Modified Files
  Find Executable Files
  Exit My Application
```

**Figure 3-2. Menu.items: Screen Output**

The next example illustrates the use of several frame descriptors in the same menu definition file to change the appearance and add to the functionality of this menu:

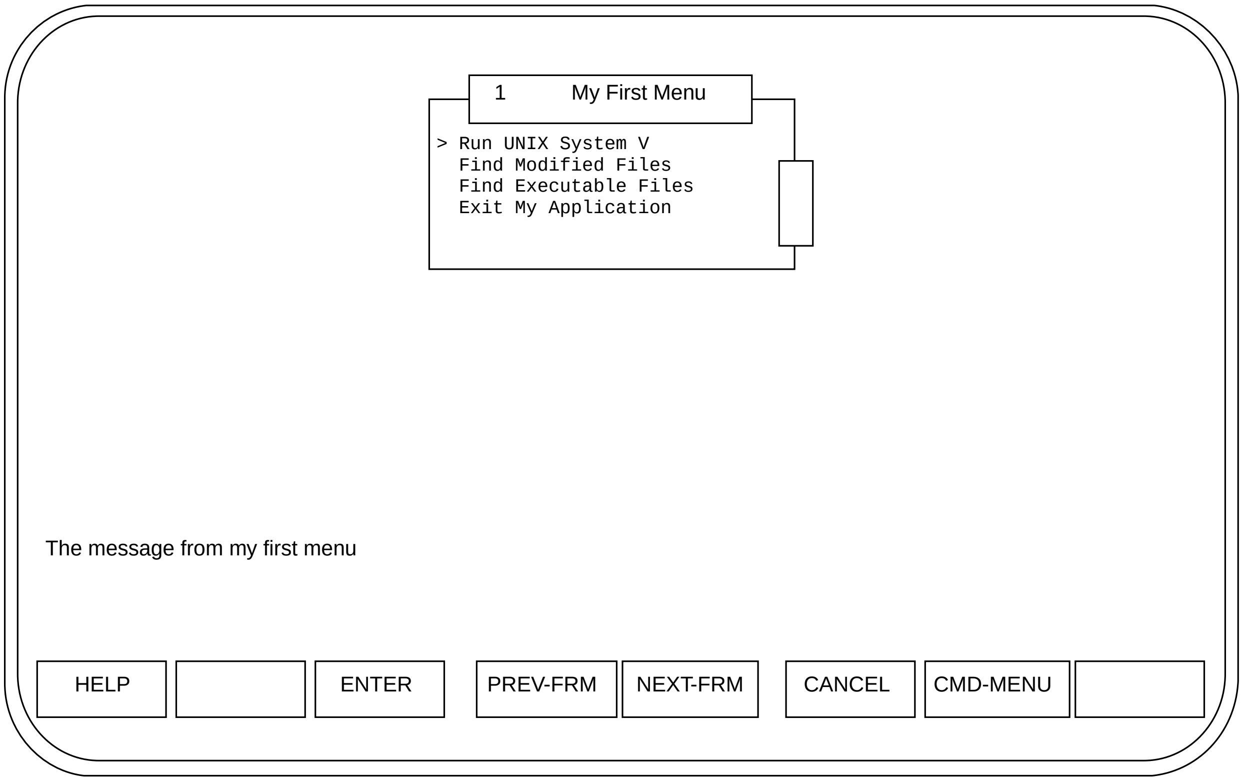
```
menu="My First Menu"
begrow=center
begcol=30
framemsg="The message from my first menu"
help=`message "A help message"`

name="Run UNIX System V"
action=unix-system

name="Find Modified Files"
action=`find $HOME -mtime -7 -print > modfiles`nop

name="Find Executable Files"
action=`find $HOME -perm -100 -print > execfiles`nop

name="Exit My Application"
action=exit
```

**Figure 3-3.  Menu.frame: An Example of Menu Frame Descriptors**

The first group of descriptors are the frame descriptors. The frame descriptor `menu` defines a title for the menu, `My First Menu`. The `begrow` and `begcol` descriptors define the position of the top left corner of the frame in the work area. In this example, the top left corner of the frame will be located at the vertical center of the work area, and column 30 horizontally.

The `framemsg` descriptor defines a string that will appear on the message line when the frame is opened. It will remain on the message line until the frame is closed or made non-current (although it can be temporarily replaced by other, shorter-term messages). The `help` descriptor defines what will happen when a user requests help while the menu is active. In this example, the string `A help message` will be displayed on the message line.

Figure 3-4 shows how `Menu.frame` will be displayed after these frame descriptors have been defined.



```
> Run UNIX System V
  Find Modified Files
  Find Executable Files
  Exit My Application
```

The message from my first menu

**Figure 3-4. Menu.frame: Screen Output**

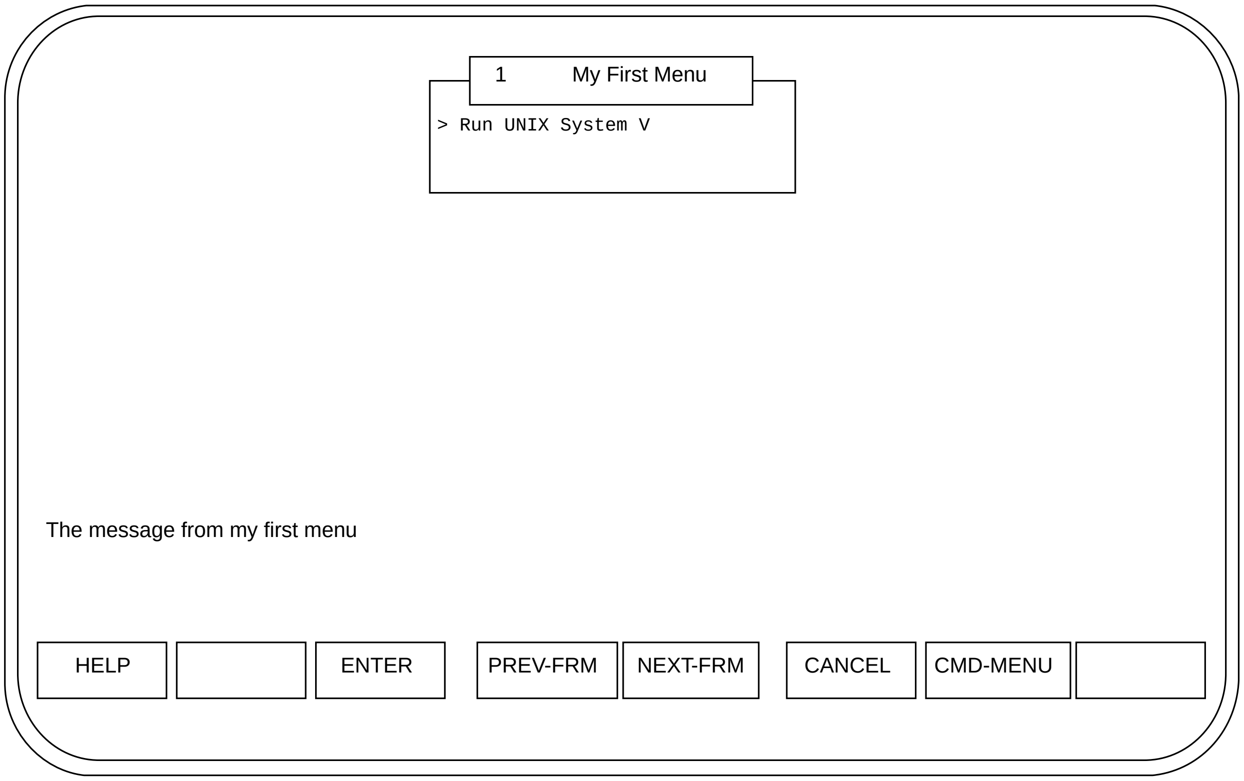
## Creating Multi-column and Scrollable Menus

The `columns` and `rows` descriptors are used to change the display of a menu. When the frame descriptor `rows` is defined in the menu definition file as follows:

```
menu="My First Menu"
begrow=center
begcol=30
rows=1
framemsg="The message from my first menu"
help=`message "A help message"`
.
.
.
```

**Figure 3-5. Menu.rows: An Example of a Scrollable Menu**

`Menu.rows` will create the following menu:



```
> Run UNIX System V
```

The message from my first menu

**Figure 3-6. Menu.rows: Screen Output**

Because the `rows` descriptor defines this menu to have only one row, only one menu item can be displayed at a time: as the user navigates to any of the other defined menu items, the menu will scroll to display it.

The same menu is displayed differently when, instead of defining the `rows` descriptor, the `columns` descriptor is defined as follows:

```
menu="My First Menu"
begrow=center
begcol=30
columns=2
framemsg="The message from my first menu"
help=`message "A help message"`
.
.
.
```

**Figure 3-7. Menu.columns: An Example of a Two-Column Menu**

The display of the menu changes as follows:

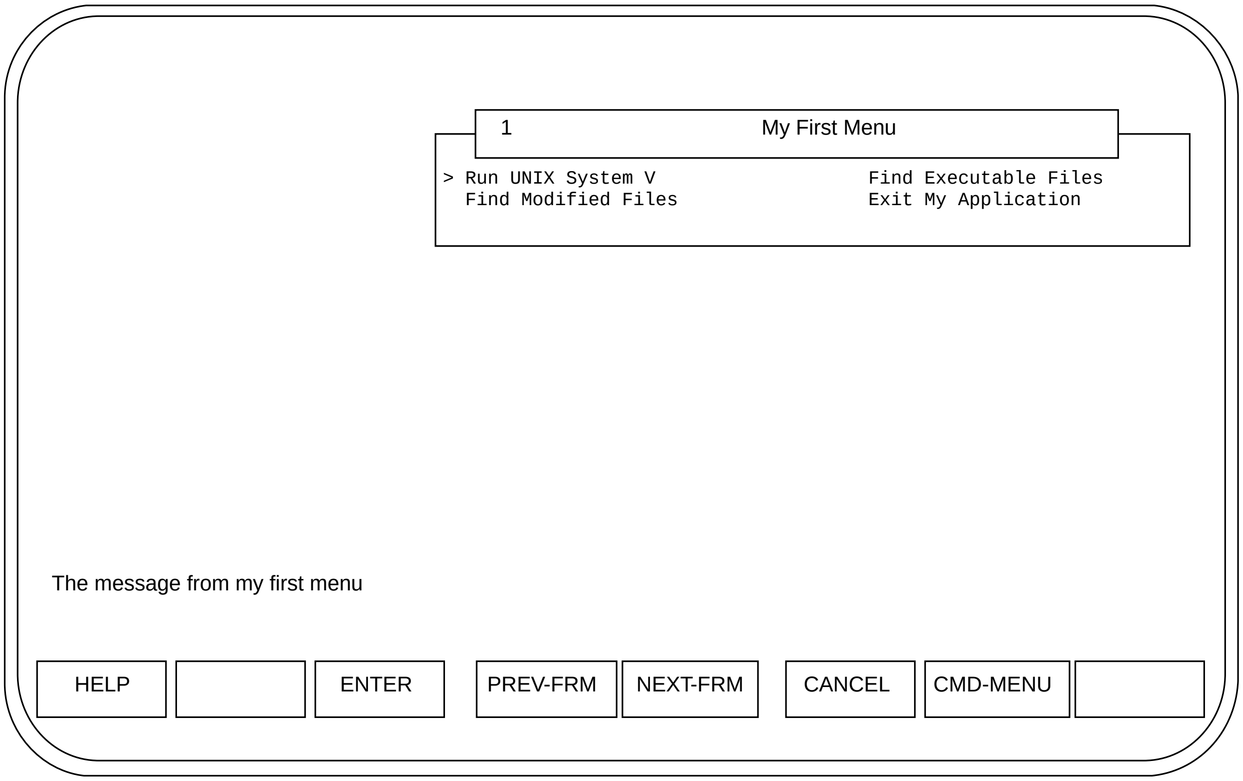

```
1                          My First Menu

> Run UNIX System V                  Find Executable Files
  Find Modified Files                Exit My Application

The message from my first menu
```

**Figure 3-8. Menu.columns: Screen Output**

The `columns` descriptor takes precedence over the `rows` descriptor if there is a conflict. For example, defining both `rows=1` and `columns=2` in this menu definition file results in the same display of the menu as shown in Figure 3-8. That is, the menu items are displayed in two columns but not one row.

## Using the reread Descriptor

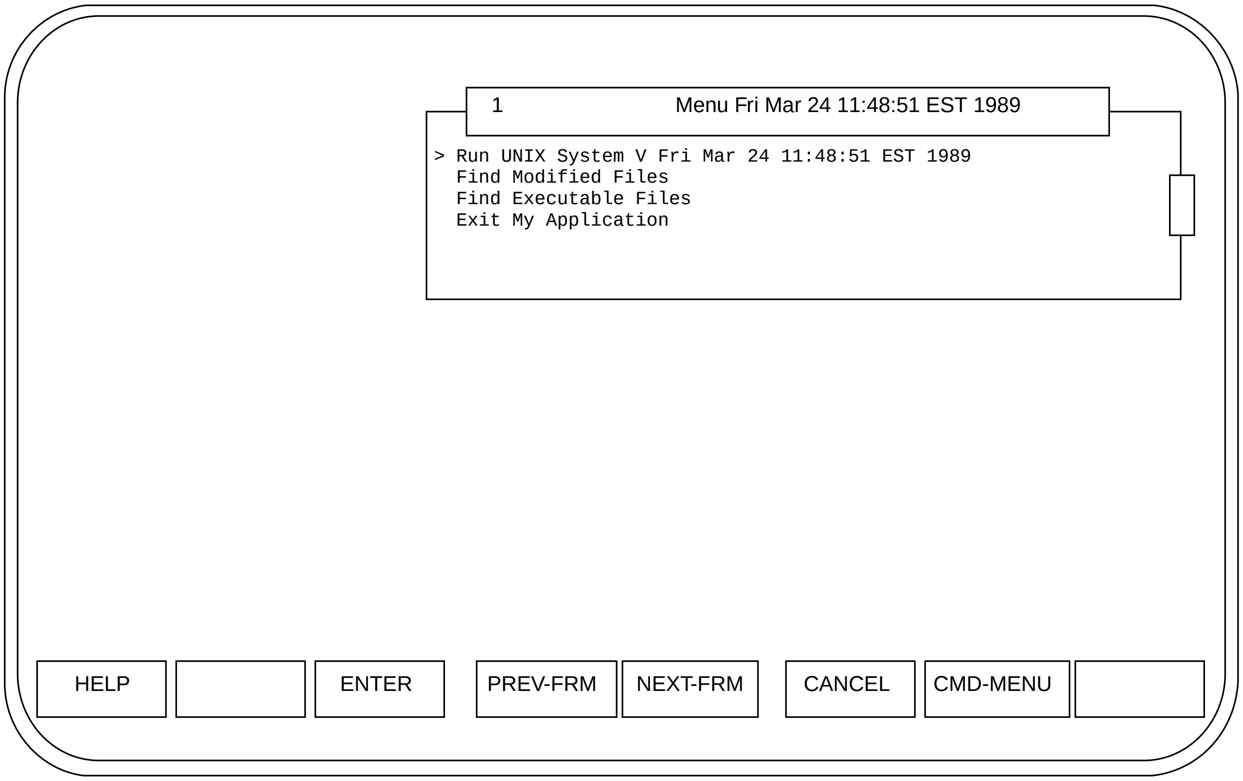
The `reread` descriptor in a frame definition file is used to request that a frame be reread when a **`checkworld`** is executed. One time that **`checkworld`** is executed is when a `SIGALRM` signal occurs (a `SIGALRM` occurs every `MAILCHECK` seconds). The frame is rebuilt if this descriptor evaluates to any value other than FALSE.

This example uses the output of the UNIX system **`date(1)`** command in the title of the menu and in the name of the first item. Recall that the `menu` descriptor is not re-evaluated when `reread` evaluates to TRUE, but the `name` descriptor is:

```
menu="Menu `date`"
begrow=center
begcol=30
reread=true

name="Run UNIX System V `date`"
action=unix-system

name="Find Modified Files"
action=`find $HOME -mtime -7 -print > modfiles`nop

name="Find Executable Files"
action=`find $HOME -perm -100 -print > execfiles`nop

name="Exit My Application"
action=exit
```

**Figure 3-9. Menu.reread: An Example of a Dynamically Updated Menu**

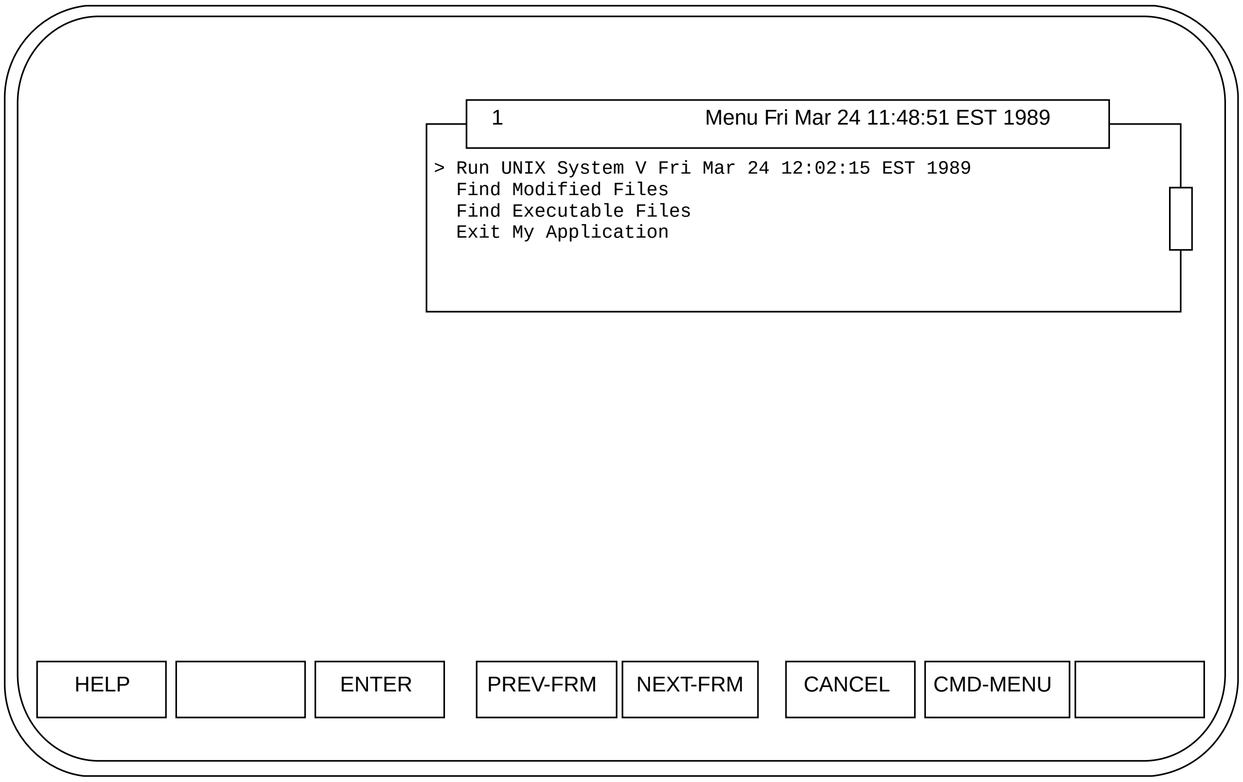
This menu definition file creates the following menu:



```
1                    Menu Fri Mar 24 11:48:51 EST 1989

> Run UNIX System V Fri Mar 24 11:48:51 EST 1989
  Find Modified Files
  Find Executable Files
  Exit My Application
```

**Figure 3-10. Menu.reread: Screen Output**

After a `SIGALRM` occurs (for example, if `MAILCHECK=180` and 3 minutes have elapsed), the date in the name of the first item changes but the date in the menu's title does not change. The menu now looks like this:



```
> Run UNIX System V Fri Mar 24 12:02:15 EST 1989
  Find Modified Files
  Find Executable Files
  Exit My Application
```

**Figure 3-11. Menu.reread: Screen Output after a SIGALRM Occurs**

## Using the interrupt and oninterrupt Descriptors

To illustrate the concepts of inheritance and scope of the `interrupt` and `oninterrupt` descriptors, the following menu definition files define them with frame descriptors and item descriptors in a menu definition file:

```
menu=“My First Menu”
begrow=center
begcol=30
interrupt=true

name=Run UNIX System V
action=unix-system

name=“Find Modified Files”
action=`find $HOME -mtime -7 -print > modfiles`nop

name=“Find Executable Files”
action=`find $HOME -perm -100 -print > execfiles`nop

name=“Exit My Application”
action=exit
```

**Figure 3-12. Menu.interrupt: An Example of Interrupt Signal Handling**

When defined among the frame descriptors, the value of `interrupt` is inherited by all processes initiated in `action` descriptors anywhere in the menu definition file, unless it is redefined for a particular item or SLK. Thus, if the user selects either the Find Modified Files or Find Executable Files item from this menu, the named process will run till its normal completion. However, the user can interrupt either process because `interrupt=true` is defined among the frame descriptors for this menu. (Note that output from Find Modified Files is saved in a file named **`modfiles`**, and that output from Find Executable Files is saved in a file named **`execfiles`**.)

When a user presses the interrupt key, the message `Operation interrupted!` appears at the bottom of the screen and the process is terminated. This is the default behavior when no other messages or actions have been defined via the `oninterrupt` descriptor, as is the case in this menu.

You can be more specific about what processes you want users to be able to interrupt and about what you want done when a process is interrupted. For example, you can block the interrupt mechanism for any item on the menu by setting the `interrupt` descriptor to FALSE for the item. A process initiated from that item cannot be interrupted, even if the frame descriptor `interrupt` is set to true:

```
menu=“My First Menu”
begrow=center
begcol=30
interrupt=true

name=“Run UNIX System V”
action=unix-system

name=“Find Modified Files”
action=`find $HOME -mtime -7 -print > modfiles`nop
interrupt=false

name=“Find Executable Files”
action=`find $HOME -perm -100 -print > execfiles`nop
oninterrupt=`message Partial output is in execfiles`nop

name=“Exit My Application”
action=exit
```

**Figure 3-13.  Menu.oninterr: A Further Example of Interrupt Handling**

If a user selects `Find Modified Files` from this menu definition file, the process it initiates cannot be interrupted. If a user selects `Find Executable Files` from this menu, it can be interrupted, and when it is, the processing defined for this item by the `oninterrupt` descriptor will occur.

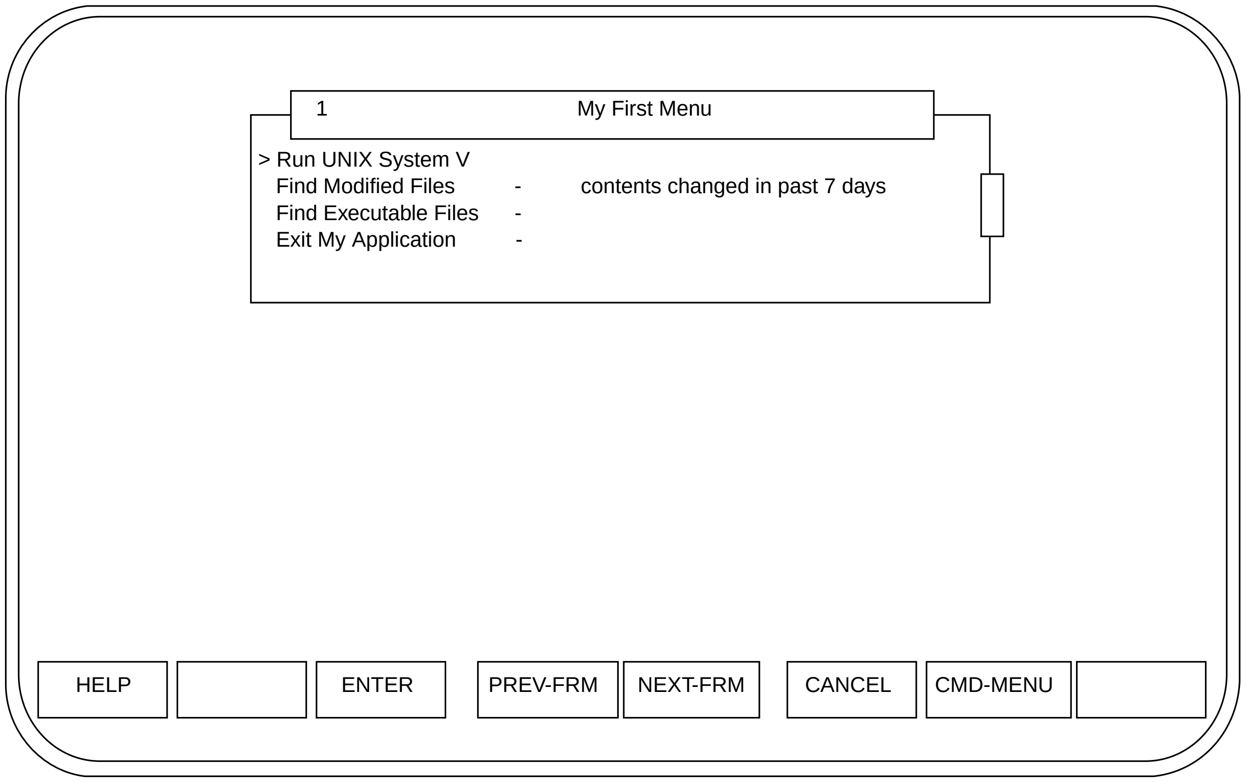
## Providing Supplementary Information for Menu Items

The item descriptor `description` defines supplementary information to be displayed on the same line as the menu item. You might want to use it to give your users a brief explanation of what an item does. This example shows how it is used.

```
menu="My First Menu"
begrow=center
begcol=30

name="Run UNIX System V"
action=unix-system

name="Find Modified Files"
action=`find $HOME -mtime -7 -print > modfiles`nop
description="contents changed in past 7 days"
interrupt=false

name="Find Executable Files"
action=`find $HOME -perm -100 -print > execfiles`nop
oninterrupt=`message Partial output is in execfiles`nop

name="Exit My Application"
action=exit
```

**Figure 3-14. Menu.descrip: An Example of the description Descriptor**

This menu definition file creates the following menu:



```
1                         My First Menu
> Run UNIX System V
  Find Modified Files     -     contents changed in past 7 days
  Find Executable Files   -
  Exit My Application     -
```

**Figure 3-15. Menu.descrip: Screen Output**

This is a single-column menu. It will be a single-column menu even if the `rows` or `columns` descriptors are defined in an attempt to make it multi-column, because when the `description` descriptor is explicitly defined, the `columns` descriptor is ignored.

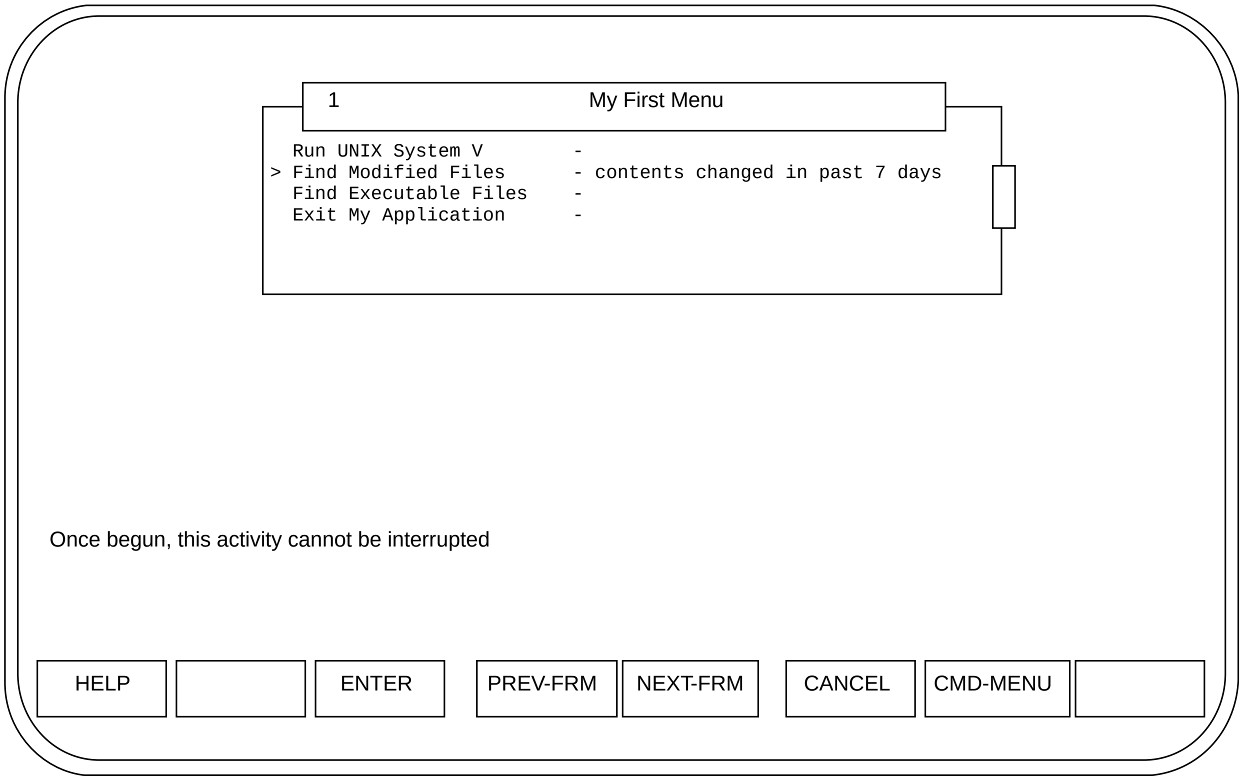
## Displaying an Item Message

This example shows how to use the `itemmsg` descriptor to display a message specific to a single menu item:

```
menu=“My First Menu”
begrow=center
begcol=30
interrupt=true

name=“Run UNIX System V”
action=unix-system

name=“Find Modified Files”
action=`find $HOME -mtime -7 -print > modfiles`nop
description=“contents changed in past 7 days”
itemmsg=“Once begun, this activity cannot be interrupted”
interrupt=false

name=“Find Executable Files”
action=`find $HOME -perm -100 -print > execfiles`nop
oninterrupt=`message Partial output is in execfiles`nop

name=“Exit My Application”
action=exit
```

**Figure 3-16. Menu.itemmsg: An Example of the itemmsg Descriptor**

Whenever the user navigates to the second menu item, the message defined in the `itemmsg` descriptor is displayed as shown in Figure 3-17. It will temporarily replace a frame message if one was created by the `framemsg` descriptor.



```
1                        My First Menu

  Run UNIX System V        -
> Find Modified Files      - contents changed in past 7 days
  Find Executable Files    -
  Exit My Application      -
```

Once begun, this activity cannot be interrupted

**Figure 3-17. Menu.itemmsg: Screen Output**

## Using the show Descriptor

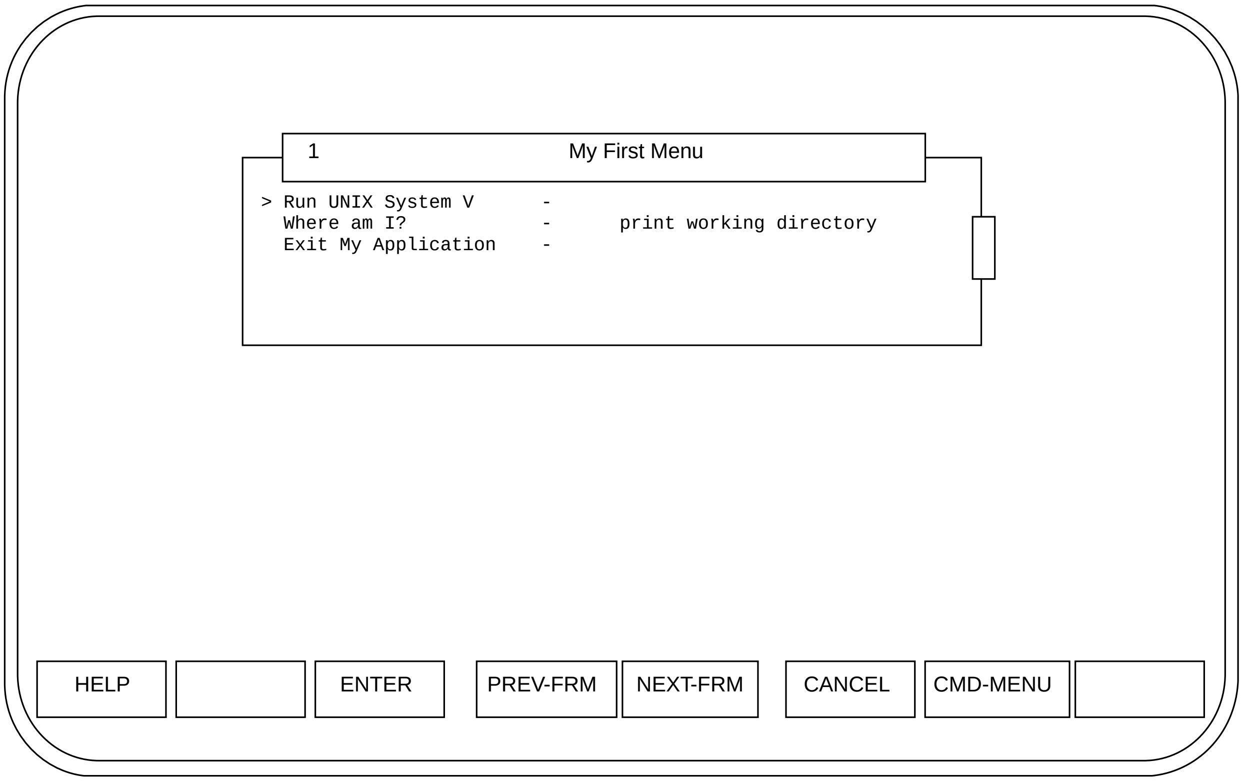
When the `show` descriptor is defined and evaluates to FALSE, the item in which it is defined does not appear on the menu. Defining the `show` descriptor as a variable allows you to decide dynamically whether or not to display a menu item.

In this example, the third item will not appear on the menu created by the following menu definition file if the login ID of the user running your application is not `root`:

```
menu="My First Menu"
begrow=center
begcol=30

name="Run UNIX System V"
action=unix-system

name="Where am I?"
description="print working directory"
action=`pwd | message`nop

name="System Administration"
action=`run sysadm`nop
show=`if [ $LOGNAME != root ];
      then echo FALSE;
      else echo TRUE;
      fi`

name="Exit My Application"
action=exit
```

**Figure 3-18. Menu.show: An Example of the show Descriptor**

This is how `Menu.show` is displayed for a user logged in as `joe`:



```
1                    My First Menu

> Run UNIX System V      -
  Where am I?            -      print working directory
  Exit My Application    -
```

**Figure 3-19.  Menu.show: Screen Output**

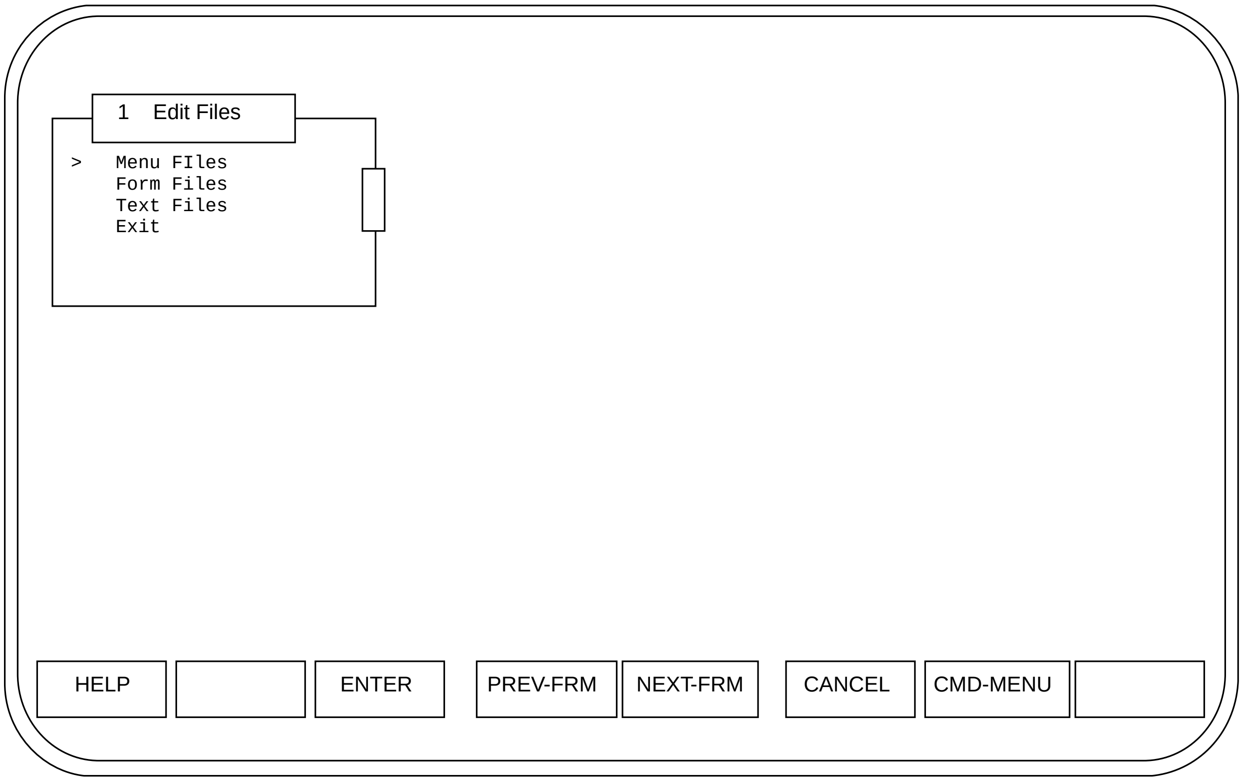
## Creating a Dynamic Menu

Now let us take a look at some more complex menus. This example shows how to pass parameters from one frame to another and how to create a menu dynamically. The example application will allow users to edit menu, form, and text frame definition files stored in the same directory. The application displays a *top menu* from which the user can select the type of frame definition file to edit:

```
menu=“Edit Files”

name=“Menu Files”
action=open Menu.dynamic Menu

name=“Form Files”
action=open Menu.dynamic Form

name=“Text Files”
action=open Menu.dynamic Text

name=“Exit”
action=exit
```

**Figure 3-20.  Menu.edit: An Example of a Dynamically Created Menu**

This menu definition file creates the following menu:



```
1   Edit Files

>   Menu FIles
    Form Files
    Text Files
    Exit
```

**Figure 3-21.  Menu.edit: Screen Output**

From this menu a user can choose to see a menu of menu frame definition files, form definition files, or text frame definition files. The action defined for all three menu items results in the same frame definition file being opened: **`Menu.dynamic.`** The last argument to the **`open`** command is a parameter that is passed to **`Menu.dynamic.`** Figure 3-22 shows the contents of **`Menu.dynamic`**:

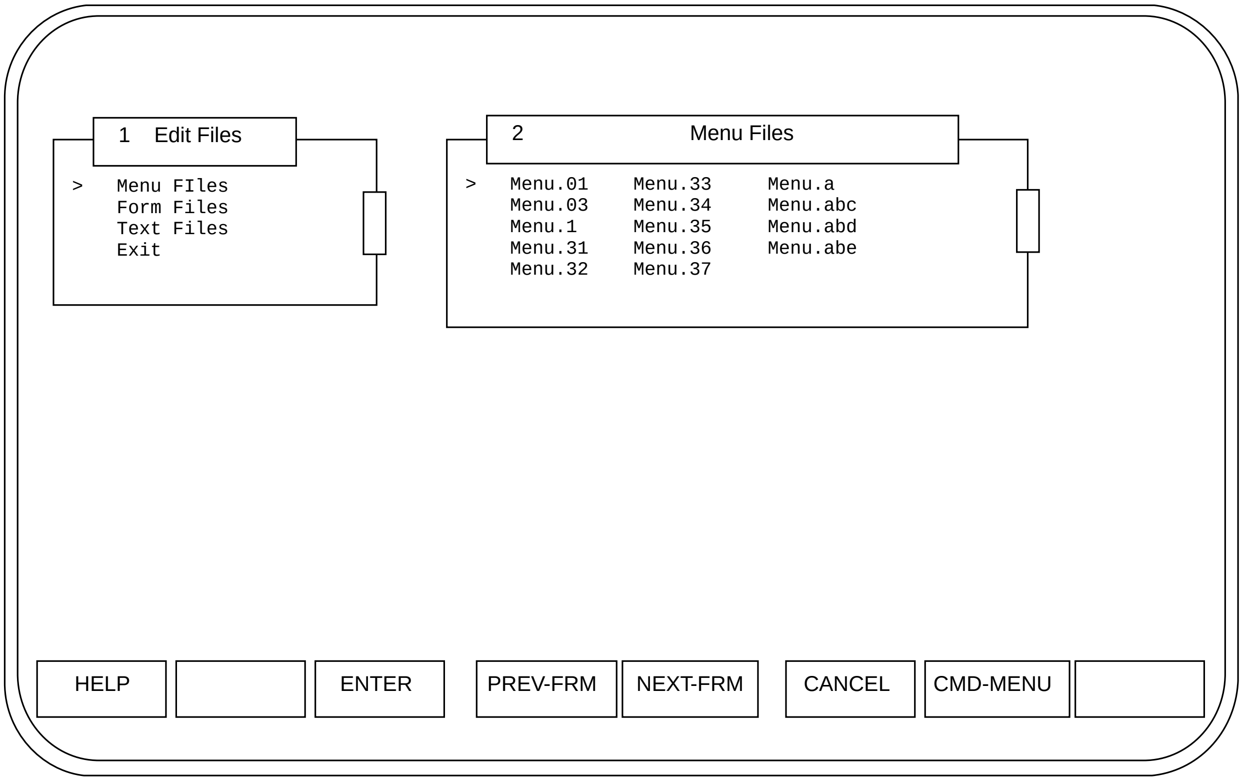
```
menu="$ARG1 Files"

`ls | regex '^('$ARG1'.*)$0$' '

name="$m0"
action=`run '$EDITOR' "$m0"`' nop`
```

**Figure 3-22. Menu.dynamic: An Example of a Dynamically Created Menu**

`Menu.dynamic` is a dynamically built menu that uses **`regex`**. The parameter passed to `Menu.dynamic` by the **`open`** command is used to build a unique title for the new menu. The items on this menu are created by a stand-alone backquoted expression. Recall that stand-alone backquoted expressions are evaluated when the frame definition file is opened, reread, or updated. That means that when `Menu.dynamic` is opened in this case, the **`ls`** command is run in the current directory, and its output is piped to the FMLI built-in utility **`regex.`**

The **`regex`** utility is used for pattern matching. Since this single **`regex`** pattern matches only files that begin with `$ARG1`, only files in the directory that have the name defined by the parameter, followed by a literal dot (.), then by any other characters, are included in the list of menu items. The menu definition file `Menu.dynamic` will create one of the three menus, based on the parameter that is passed to it. The items in the menu it creates are based on the files whose names match the pattern being searched for by the **`regex`** utility. The menu item template defined in this menu definition file provides that the `name` descriptor has the value `$m0,` which evaluates to the pattern enclosed in parentheses (a file name) in the **`regex`** expression. The template provides that the `action` descriptor will invoke the defined editor on the file. All file names matched by the **`regex`** utility are passed to this menu item template, FMLI determines the size and shape of the menu frame based on the total number of items produced, and the menu frame is posted.

For example, when `Menu Files` is selected from the menu titled `Edit Files`, `Menu.dynamic` receives the parameter `Menu`, **`regex`** searches for file names beginning with **`Menu.`** ("menu dot"), and displays the following second menu:



```
1  Edit Files                       2                 Menu Files
>  Menu FIles                       >  Menu.01    Menu.33    Menu.a
   Form Files                          Menu.03    Menu.34    Menu.abc
   Text Files                          Menu.1     Menu.35    Menu.abd
   Exit                                Menu.31    Menu.36    Menu.abe
                                       Menu.32    Menu.37

HELP         ENTER    PREV-FRM  NEXT-FRM   CANCEL  CMD-MENU
```

**Figure 3-23.  Menu.edit: Screen Output when Menu Files Is Selected**

By selecting the appropriate item from the `Menu Files` menu, the user is able to edit any of the menu, form, or text frame definition files in the directory. If one of these types of frame definition files is not present in the  directory, the corresponding menu is not created.

## Form Frame Descriptors

A form definition file can begin with an optional set of frame descriptors, followed by one or more sets of field descriptors (one set per field), and it can end with one or more optional sets of descriptors that define the screen-labeled function keys that will be displayed when the form is the active frame in the user's work area (one set per SLK).

Some of the attributes of a form that you can define are the following:

- the title of the form
- the screen position of the form
- a label and input area for each field

- any initial value to display for each field
- a set of choices for the value of a field
- the starting position and length of each field and label
- the validation to be done for each field, and the error message to display if validation fails
- whether the form is multi-page or not
- labels and functions for the SLKs of the form

The descriptors in a form definition file must be in the following order:

```
[frame_descriptor_1
.
.
.
frame_descriptor_n]

field-one_descriptor_1
.
.
.
field-one_descriptor_n

[field-two_descriptor_1
.
.
.
field-two_descriptor_n]

. . .

[SLK-n_descriptor_1
.
.
.
SLK-n_descriptor_n

. . .]
```

**NOTE**

Out-of-order descriptors will be ignored if this order—frame, then fields, then SLKs—is not followed.

## Frame Descriptors for Forms

The optional set of frame descriptors can be any valid frame descriptors for forms, in any order. If a descriptor appears more than once in the set, the last one is used.

**NOTE**

Although technically none of the frame descriptors is required in a form definition file, a form that does not define the `done` descriptor is virtually useless, since no user input will be recorded when the `SAVE` key is pressed.

| | |
|---|---|
| `altslks` | The `altslks` descriptor defines whether SLKs 9 through 16 are displayed when the frame is initially opened. If `altslks` evaluates to TRUE, SLKs 9 through 16 will be displayed.<br><br>The default, if this descriptor is not defined, is FALSE, which causes SLKs 1 through 8 to be displayed. |
| `autolayout` | The `autolayout` descriptor specifies whether FMLI should use reasonable defaults for the `fcol`, `frow`, `ncol`, `nrow`, and `columns` field descriptors in this form. If `autolayout` evaluates to TRUE, the reasonable defaults will be used; if it evaluates to FALSE, the defaults for these 5 field descriptors will be -1. The default, if this descriptor is not defined, is FALSE.<br><br>Using this descriptor and default field descriptor values allows forms to be created more easily, since when it evaluates to TRUE, the only required descriptor to define a field is the `name` descriptor, specifying the field label. Without this descriptor, all fields require 6 descriptors to be defined. More information may be found in “Automatic Layout of Form Fields” on page 3-40. |
| `begrow`, `begcol` | The `begrow` and `begcol` descriptors define the original position of the top left corner of the form frame in the user's work area. (`begrow=0` and `begcol=0` evaluates to the upper left corner of the work area.) When writing international applications, `autosort` should always be set to TRUE, because the fields used in a form will have different lengths in different languages. These descriptors accept values of type position: |

| | |
|---|---|
| `center` | the form frame will be centered in the work area |
| `current` | the form frame overlaps the current frame's position (valid for `begrow` only) |
| `distinct` | the form frame will not overlap the current frame if possible (valid for `begrow` only) |
| `any` | FMLI chooses a position with least amount of total overlap |
| *integer* | the form frame will be positioned in an absolute position, defined by *integer.* Defining `begrow` and `begcol` to be integer values causes the frame to appear in the given position. |

If either `begrow` or `begcol` evaluates to `center`, then the other can only be an integer value or `center`. Any other value is ignored and the descriptor defaults to `center`.

If neither is `center`, then the value of `begrow` determines the value of `begcol`: if `begrow` is `current`, `distinct`, `any`, or an invalid value, then `begcol` defaults to `any`. If `begrow` is a valid integer, `begcol` can be a valid integer; if `begcol` is an invalid integer in this case, it defaults to `any`. If integer values are supplied and either `begrow` or `begcol` are outside the screen boundary, a default value of `any` will be used.

| | |
|---|---|
| `close` | The `close` descriptor is evaluated when the form is closed and when the user exits from the FMLI application. The `close` descriptor is of type null, which means its only purpose is to obtain the side effects of backquoted expressions coded in its definition. |
| `done` | The `done` descriptor defines the command to be executed when the user selects `SAVE`. If `done` is not defined, it defaults to the FMLI command **`close`**. Note that user input is not saved automatically; user input to the form should be recorded by backquoted expressions in the `done` descriptor. |
| `framemsg` | The `framemsg` descriptor displays its value on the message line for as long as the frame is current. It can be temporarily replaced by a message displayed when: • the `fieldmsg` descriptor is defined for a specific field in the form frame • a message is generated by the `message` built-in utility with the **`-t`** (default) option • an FMLI error message is generated. It can be replaced for as long as the frame is current by a message generated by `message` with the **`-f`** option. (See the **`message(1F)`** manual page.) |
| `form` | The `form` descriptor defines the title of the form frame. It will be truncated if it is longer than `DISPLAYW-6`. |
| `help` | The `help` descriptor specifies what will happen when the user requests help while in the form. Since this descriptor is evaluated at the time the user requests help, the specification of what help is displayed can be determined through parameters that are set interactively. |
| `init` | The `init` descriptor defines whether the form frame will be opened. If this descriptor is not defined, it defaults to TRUE, which means the form frame will be opened. If `init` evaluates to FALSE, the form frame will not be opened. If `init` evaluates to FALSE on an update, the frame is closed, unless it is an initial frame. |
| `interrupt` | The Boolean descriptor `interrupt` defines whether an executable that is coded in `action` or `done` descriptors can be inter- |

rupted by users (FALSE means not interruptible, TRUE means interruptible). It is subject to an inheritance hierarchy: if not defined anywhere in your application, the default value FALSE applies throughout. If explicitly defined at any inheritance level, then executables in `action` and `done` descriptors at or above that inheritance level will inherit that defined value. (See “Interrupt Signal Handling” on page 2-39 for complete information.)

For example, if defined among the frame descriptors in a form definition file, the value of `interrupt` is inherited by the frame descriptor `done`, and the `action` descriptor in all sets of SLK descriptors in the form, unless it is redefined for a specific SLK.

`lifetime`

The `lifetime` descriptor defines when the form will be closed (that is, removed from the work area). It is evaluated whenever the form is opened, closed, made current, or made non-current. The acceptable values are:

| | |
|---|---|
| `shortterm` | the form closes whenever the user navigates to another frame or when the command line is accessed (the user presses `CTRL-j` or `CTRL-f c`) |
| `longterm` | the form closes when the user issues a **`cleanup`** or **`close`** command |
| `permanent` | the form closes whenever the user issues a **`close`** command |
| `immortal` | the form closes only when the user exits from the application |

The `lifetime` descriptor is ignored in form definition files that are given as arguments when **`fmli`** is invoked: such forms have a lifetime of `immortal`. See “Other Useful Examples” on page 3-61 for an example of how this descriptor may be used to close a form when another frame is opened or updated.

`oninterrupt`

The command descriptor `oninterrupt` defines what will happen when an interrupt signal is received. If `interrupt` is not coded anywhere in your application, or if it evaluates to FALSE, `oninterrupt` is ignored.

`oninterrupt` is subject to an inheritance hierarchy: if not defined anywhere in your application, the default value `` `message Operation interrupted!` nop `` applies throughout. If explicitly defined at any inheritance level, then executables in `action` and `done` descriptors at or above that inheritance level will inherit that defined value. (See “Interrupt Signal Handling” on page 2-39 for complete information.)

For example, if defined with the frame descriptors in a form definition file, the value of `oninterrupt` is inherited by the frame descriptor `done`, and by the `action` descriptor in all sets of SLK descriptors, unless redefined for a specific SLK.

| | |
|---|---|
| `reread` | If `reread` is not defined, it defaults to FALSE. If `reread` evaluates to TRUE, the form will be periodically updated by rereading its description file when the **`checkworld`** command is executed. **`checkworld`** is executed when a `SIGALRM` alarm occurs (every `$MAILCHECK` seconds). Other times **`checkworld`** is executed include when a frame is opened, closed, or navigated to. (See **`checkworld`** in "Built-in Variables" on page 2-7.) When **`checkworld`** occurs, all frames whose `reread` descriptor evaluates to TRUE will be updated. (However, the `form` descriptor is not reread.) Execution of **`checkworld`** may cause the message line to clear. |

## Field Descriptors

The following descriptors can be defined once for each field in a form. In each set of field descriptors, `name` must be first. If a descriptor appears more than once in a set, the last one is used.

**NOTE**

There must be at least one active, visible (that is, `show=true`) field in a form. If you open a form and none of the fields can be posted because `rows` or `columns` is negative or `0`, or if `frow` or `fcol` is negative, FMLI will display an empty frame with the cursor positioned in the title bar. If the form definition file contains properly defined field labels, they will be displayed.

| | |
|---|---|
| `autoadvance` | The `autoadvance` descriptor defines whether a RETURN is automatically performed when a user fills in the last character of a field. It defaults to FALSE (an automatic RETURN is not performed).<br><br>If `autoadvance` is defined and evaluates to TRUE, when the user types in `columns` characters, the field will be automatically validated, and if valid, the cursor will be automatically advanced to the next field. This descriptor is ignored in vertically or horizontally scrollable fields. |
| `choicemsg` | The `choicemsg` descriptor defines information to be displayed on the message line when the user presses CHOICES. The `choicemsg` descriptor displays a message with transient duration. That is, it remains on the message line only until the user presses another key or a **`checkworld`** occurs. Transient messages take precedence over frame duration messages and permanent duration messages (see the **`message(1F)`** manual page for more information). If `choicemsg` is not defined, the default is `There are no choices available`. |
| `columns` | See the entry for `rows`, `columns` later in this section. |

fieldmsg

The fieldmsg descriptor defines information that will appear on the message line when this field is navigated to. The fieldmsg descriptor displays a message with transient duration. That is, it remains on the message line only until the user presses another key or a **checkworld** occurs. Transient messages take precedence over frame duration messages and permanent duration messages (see the **message(1F)** manual page for more information).

frow, fcol

The frow and fcol descriptors define the position of the field input area in the frame. The value of frow can be an integer greater than or equal to 0 and less than DISPLAYH-2; the value of fcol can be an integer greater than or equal to 0 and less than DISPLAYW-2. (frow=0 and fcol=0 evaluate to the upper left corner of the frame, that is, to the first available row and column, respectively.) If either value is negative or if either value is too large (the position is off the screen), then the field input area will not be displayed. If no field input areas are displayed on a page of a form, the cursor is positioned in the title bar.

When autolayout evaluates to FALSE, these descriptors default to -1.

When autolayout evaluates to TRUE, frow defaults to the value *current_nrow*, where *current_nrow* is the value of nrow for the field being defined. That is, by default the row in which the field input area appears will be the same as the row in which the label of the field appears.

When autolayout evaluates to TRUE, fcol defaults to the greater of

- the value *previous_fcol*, where *previous_fcol* is the value of fcol for the previous field in the form; or
- the value 1+*current_ncol*+*lengthOfLabel*, where *lengthOfLabel* is the number of characters contained in the label specified with name for the field and *current_ncol* is the value of the ncol descriptor for the field.

If the field is the first field in the form, fcol defaults to 1+*current_ncol*+*lengthOfLabel*. By default, then, the input area of any field whose label is as long or longer than that of a previous field will be separated from the field label by one space. (Recall that position 0 is the first available column.) You can code the name descriptor for the first field with padded blanks to cause the value of *lengthOfLabel* to be longer than the actual word length of the label. The name and padded blanks must be surrounded by matching single or double quotes. That is, name='Field1' would have a length of 9, not 6, for the purpose of default positioning of the field. Assuming ncol has a value for 0 for the field, the value of fcol would be 10. Since the field will start at position 0, that will leave four spaces between the label and the input area.

`inactive` The `inactive` descriptor defines a form field that is displayed in the form, but cannot be navigated to. If this descriptor is not defined, it defaults to FALSE (the field will be active). If this descriptor evaluates to TRUE, the field is displayed in the form, without an underline, and cannot be navigated to. By default, the `inactive` descriptor is evaluated when the form is opened and thereafter whenever navigation occurs from a field whose value has been changed. There must be at least one active field for a form to be displayed.

`invalidmsg`, `invalidOnDoneMsg`

The `invalidmsg` descriptor is used with the `valid` or `menuonly` descriptors and defines a string that will be printed on the message line when the input for the field is invalid. The default is `Input is not valid`.

The `invalidOnDoneMsg` descriptor is used with the `validOnDone` descriptor and defines a string that will be printed on the message line when `validOnDone` evaluates to FALSE. The default is `Relationship of values in 2 or more fields is not valid`. If you use a different message, make sure it indicates to the user that more than one field is involved.

These descriptors display a message with transient duration. That is, it remains on the message line only until the user presses another key or a **`checkworld`** occurs. Transient messages take precedence over frame duration messages and permanent duration messages (see the **`message(1F)`** manual page for more information).

`lininfo` The `lininfo` descriptor defines a string that will be assigned to the built-in variable `LININFO` when the user navigates to the field. If `lininfo` is not defined, `LININFO` evaluates to the null string.

`menuonly` The `menuonly` descriptor defines the choices listed in the `rmenu` descriptor to be the only valid input for the field. If this descriptor evaluates to TRUE, then the user must input one of the choices in `rmenu` for the field. `menuonly` must only be used when the `rmenu` descriptor has been defined using the curly brace format; otherwise no input will be valid.

**NOTE**

If you define `menuonly` for a field, do not define the `valid` descriptor.

`name` The `name` descriptor defines the label of the field. The value you define for `name` should tell the user what piece of information is wanted in the field.

It can also be used to display a label, such as `Page 2 of 5`, if it and these other descriptors for the field are defined as follows:

- the `name` descriptor is defined as the desired label—in this case `Page 2 of 5`
- the `page` descriptor is set to the appropriate value—in this case `2`
- the `inactive` descriptor evaluates to TRUE
- the `columns` descriptor evaluates to `0`, so the input area does not display

`noecho`

The `noecho` descriptor defines whether what the user enters in the field input area will be displayed. This descriptor defaults to FALSE (input will be displayed). If `noecho` does not evaluate to FALSE, then what the user enters in the field will not be echoed in the display (this descriptor is often used when the requested input is a password). `noecho` should not be defined for multi-line fields.

`nrow`, `ncol`

The `nrow` and `ncol` descriptors define the position of the first character of `name` in the form. These descriptors accept an integer value: the value of `nrow` can be greater than or equal to `0` and less than `DISPLAYH-2`; the value of `ncol` can be greater than or equal to `0` and less than `DISPLAYW-4`. (`nrow=0` and `ncol=0` evaluate to the upper left corner of the frame, that is, to the first available row and column, respectively.) If either value is negative, `name` will not be displayed. If either integer is too large (the position is off the screen), the entire form will not be displayed.

When `autolayout` evaluates to FALSE, these descriptors default to -1.

When `autolayout` evaluates to TRUE, `nrow` defaults to the value *previous_nrow*+*previous_rows*, where *previous_nrow* and *previous_rows* are the values, respectively, of `nrow` and `rows` for the previous field in the form. If the field is the first field in the form, or the first field on the page of a multi-page form, the default is 0. By default, then, unless the field is the first one on a page, its label will appear one row below the last row of the previous field.

When `autolayout` evaluates to TRUE, `ncol` defaults to its value in the previous field, or 0 if the field is the first field of the form.

`page`

The `page` descriptor allows you to define the page of a form on which the field will appear. It accepts integer values greater than `0`. A value of 0 or a negative value will cause the field not to appear in the form.

`page` defaults to 1 (the field will appear on the first page of the form). A value greater than 1 creates a multi-page form. That is, if `page` is defined for a field and evaluates to `2`, for example, the field will appear on the second page of the form.

`rmenu` The `rmenu` descriptor defines a list of choices for a field. Two formats are acceptable when defining this descriptor:

- The first format is a list of choices, separated by white space, enclosed in braces. The white space after the opening brace and before the closing one is mandatory:

```
rmenu={ item1 item2 item3 ... itemn }
```

**NOTE**

If your definition of `rmenu` degenerates to an empty list, `rmenu={}`, the value of `choicemsg` will be displayed: your definition, if any, or the FMLI default message `There are no choices available`. If you define `choicemsg`, be sure it is appropriate to the “empty list” case.

By default, if this list has three or fewer items, the choices are displayed in the field itself. The first item appears when the user presses the `CHOICES` SLK; the user toggles through the choices by pressing the same key. If the list has more items, the choices will appear in a pop-up choices menu. (See Chapter 4 for a discussion of the `toggle` descriptor, which can be used to change this default behavior.) When the user selects an item in a pop-up choices menu, the selection is automatically placed in the built-in variable `Form_Choice`, the value of which is entered in the field when the choices menu closes.

- The second format is an **`open`** command. When you use this format, the `rmenu` descriptor evaluates to opening a menu, and the user selects from that menu. For example:

```
rmenu=open Menu.mtgdates
```

The action associated with each choice in the menu must set the built-in variable `Form_Choice` and close the menu. When the user selects an item in the menu, the selection is placed in `Form_Choice`, the value of which is entered in the field when the choices menu closes. The `toggle` descriptor is ignored when this format is used—a pop-up choices menu is always displayed.

**NOTE**

Do not use `menuonly` with this format. To validate the choices in the menu, use the `valid` descriptor.

`rows`, `columns`

The `rows` and `columns` descriptors define the size of the input area, the length and width, respectively, of the region in which the user can enter input. These descriptors accept an integer value: the value of `rows` must be greater than `0` and less than `DISPLAYH-2`; the value of `columns` must be greater than `0` and less than `DISPLAYW-4`. If either is less than or equal to `0`, the field will not be displayed. If either value is too large (the position is off the screen), the entire form will not be displayed.

`rows` defaults to 1 (the field will be one row long). A value greater than `1` creates a multi-line field.

If `autolayout` evaluates to FALSE, `columns` defaults to -1. If `autolayout` evaluates to TRUE, `columns` defaults to its value in the previous field, or 4 (the field will be four columns wide) if the field is the first field of the form. Although a default value cannot be picked that will be useful for real applications, a default value can be useful for learning purposes and for writing test scripts; hence the default 4.

`scroll`

The `scroll` descriptor defines whether the field input area can scroll. There are two types of scrolling: vertical, for multi-line fields; and horizontal, for single-line fields. If not defined, this descriptor defaults to FALSE (field input area cannot scroll).

If `scroll` evaluates to TRUE, then the field input area can be scrolled. This means that the field input area can be as long as the entry the user types, and the `columns` descriptor is not a limit for the length of user input. For single-line fields, the last space in `columns` is reserved for scroll symbols: `>` means the field can be scrolled to the right, `<` means the field can be scrolled to the left, and `=` means the field can be scrolled either left or right. For example, after the user types in `columns-1` valid characters, the field will scroll, and the `<` symbol will appear in the last space, indicating that the input in the field has scrolled to the left. For multi-line fields, scroll indicators for up (`^`) and down (`v`) appear in the bottom right border of the frame when the user has entered data up to the last character in the last displayed line of the field. For example, after the user types in `rows` lines of information, the scroll indicator for up (`^`) appears in the lower right border.

`show`

The `show` descriptor defines whether a field will be displayed in the form. If `show` evaluates to FALSE, then the label and input area will not be shown. There must be at least one field for which `show` evaluates to TRUE, or the form will not open. By default, the `show` descriptor is evaluated when the form is opened and thereafter whenever navigation occurs from a field whose value has been changed. Note that even if the field is not shown, it still counts as a field for the purpose of evaluating the built-in variable `F`*n*. (See “Variables” on page 2-6 for more information about the built-in variable `F`*n*.)

`valid`

The `valid` descriptor defines whether the input to a field is valid. If `valid` evaluates to FALSE, the current input is considered invalid and FMLI will not process the field or evaluate the `done` descriptor. Checking the validity of the field is often done by evaluating a backquoted expression. The backquoted expression must be coded to evaluate to TRUE when the value is valid, FALSE otherwise.

**NOTE**

The built-in utility **`regex`** is often used in a `valid` descriptor for field validation. For example, it can be used to require that part of a field be non-numeric.

The FMLI conditional statement has essentially the same functionality as the UNIX system shell conditional statement, and can be used to do more complicated validations.

Before a user leaves a form, each field that defines the `valid` descriptor is validated at least once, at one of the following times. Note that the critical factors are whether the field was modified, and which key was used to navigate away from it.

1. If the field has been visited and modified, validation occurs upon any navigation key being pressed.
2. If the field has been visited but not modified, validation occurs upon the ENTER key being pressed. The use of other navigation keys will not initiate validation.

   (Note that this behavior is new in FMLI 4.0+. Thus, users can now navigate to other fields when, for example, leaving the current field blank would cause it to fail a validation test.)
3. If the field has not been visited, or if it was visited but not modified and some key other than ENTER was used to navigate away from it, validation occurs upon the SAVE key being pressed.

So, if a user opens a form frame with five fields that define the `valid` descriptor, but only modifies the first two fields, the two modified fields are validated before the use can leave them. The remaining three fields will be validated when the user saves the form.

If any field (including fields on other pages in a multi-page form) does not pass its validation test, FMLI will not process the field, the cursor will jump to the invalid field if not already on it, the message defined in the descriptor `invalidmsg` (if defined, otherwise the default message) will be displayed on the message line, and the `done` descriptor will not be evaluated. In the case of more than one invalid field, this behavior will be repeated in field order, each time the SAVE SLK is pressed.

For the reasons given in the validOnDone entry below, you should use valid only to validate the value of a single field, without reference to other fields. Use validOnDone to validate the relationship between the values of different fields.

You should be cautious using the valid descriptor for a field that could become inactive. Unexpected behavior can occur.

validOnDone

The validOnDone descriptor does the same thing as the valid descriptor but is evaluated only when the user attempts to save the form, and is used with the invalidOnDoneMsg descriptor rather than the invalidmsg descriptor. You should use it to validate the relationship between the values of different fields, as in the following scenario.

Suppose you have defined Major, Degree, and College fields in a form, in that order. For the Degree field you have defined a validation test that will disallow the values BS or MS when Major has the value History. If you use valid to perform the test

```
valid=`test ( “$F1” = “History” -a
                  ( “$F2” = “BA” -o “$F2” = “MA” )
            ) -o
            ( “$F1” = “Electrical Engineering” -a
                  ( “$F2” = “BS” -o “$F2” = “MS” )
            ) `
invalidmsg=vary The “$F2” degree is not offered in “$F1”
```

FMLI will correctly disallow an entry of BS in Degree when Major is History.

Suppose now, though, that the user has entered Electrical Engineering in Major and BS, a valid value, in Degree. This user has a change of mind and, after navigating back to Major, changes its value to History. Because valid has already been evaluated for Degree, FMLI will not check its value against the new value of Major unless Degree is revisited. It will check it, however, if the validOnDone descriptor is coded for Degree

```
validOnDone=`test ( “$F1” = “History” -a
                        ( “$F2” = “BA” -o “$F2” = “MA” )
                  ) -o
                  ( “$F1” = “Electrical Engineering” -a
                        ( “$F2” = “BS” -o “$F2” = “MS” )
                  ) `
invalidOnDoneMsg=vary The “$F2” degree is not offered in “$F1”
```

because `validOnDone` is evaluated when the user attempts to save the form. You would use `validOnDone` in a similar way if, say, you wanted to disallow the value `Business` for `College` when `Major` had the value `History` and `Degree` had the value `BA`.

When `validOnDone` evaluates to FALSE for a field, the cursor is positioned in the input area of that field. Because the descriptor is evaluated only when the user attempts to save the form, the user can navigate away from the field to any other field. In other words, the user of the example application above could navigate away from `Degree` to `Major` and change its value to `Electrical Engineering`. As this implies, all `validOnDone` descriptors for a form will be re-evaluated each time the user attempts to save the form (because the new value of `Major`, although valid in relation to the value of `Degree`, may now be invalid in relation to the value of `College`).

**NOTE**

The `validOnDone` descriptor can be coded as `validOnDone=validonentry` to request that any validation done with the `valid` descriptor be repeated when the user attempts to save the form. This allows you to code one validation in two places, eliminating a possible maintenance problem.

`value` The `value` descriptor defines the default value for the input field. If this descriptor is defined, its value will be displayed in the field when the form is opened or updated. The default is not changed by the user entering data into the field. That is, the default value is restored when the form is opened or updated, or if the built-in utility `reset` is run while the field is current. Note that `valid` or `validOnDone` can be used to validate the input to fields defined to have default values.

`wrap` The `wrap` descriptor defines whether word wrap will occur if a word will not fit on the current line of a multi-line field. If this descriptor is not defined, it defaults to FALSE. If `wrap` evaluates to FALSE, then the cursor will not automatically wrap to the next input line. If `wrap` evaluates to TRUE, and *word* will not fit on the current line but will fit on the next line, then *word* will automatically be moved to the next line. The `wrap` descriptor is ignored in a single-line field.

**NOTE**

Screen labels and actions for function keys can be defined in a form description file as well as at the initialization file level. Each set of screen-labeled function key descriptors must include the `name` and `button` descriptors; the `name` descriptor must be first. If a descriptor appears more than once in a set, the last one is used. See Chapter 4 for a discussion of how to use the screen-labeled function key descriptors.

## Automatic Layout of Form Fields

In previous versions of FMLI the descriptors that define the position and size of fields and their labels (`frow`, `fcol`, `nrow`, `ncol`, and `columns`), defaulted to -1. In effect, this meant that those five descriptors had to be defined for each field in the form.

With FMLI Release 4.0+, new, more reasonable defaults are available for these descriptors. However, to preserve compatibility with older applications, the new defaults are only available if the `autolayout` frame descriptor for the form evaluates to TRUE, or if the application descriptor `autolayout` evaluates to TRUE and the same descriptor for the form is not coded. When the `autolayout` descriptor is defined, the new defaults provide automatic layout of the fields and labels of a form; the only required field descriptor is `name`. The new defaults are described in detail in “Field Descriptors” on page 3-31.

The defaults enabled by `autolayout` allow easier coding for simple forms and test scripts, rapid prototyping, and provide a reasonable default form appearance, although `nrow`, `ncol`, `frow`, `fcol`, and `columns` can still be used to obtain precisely formatted forms. In addition, it is still possible, using the previous defaults of -1 for these descriptors, to obtain the following refinements:

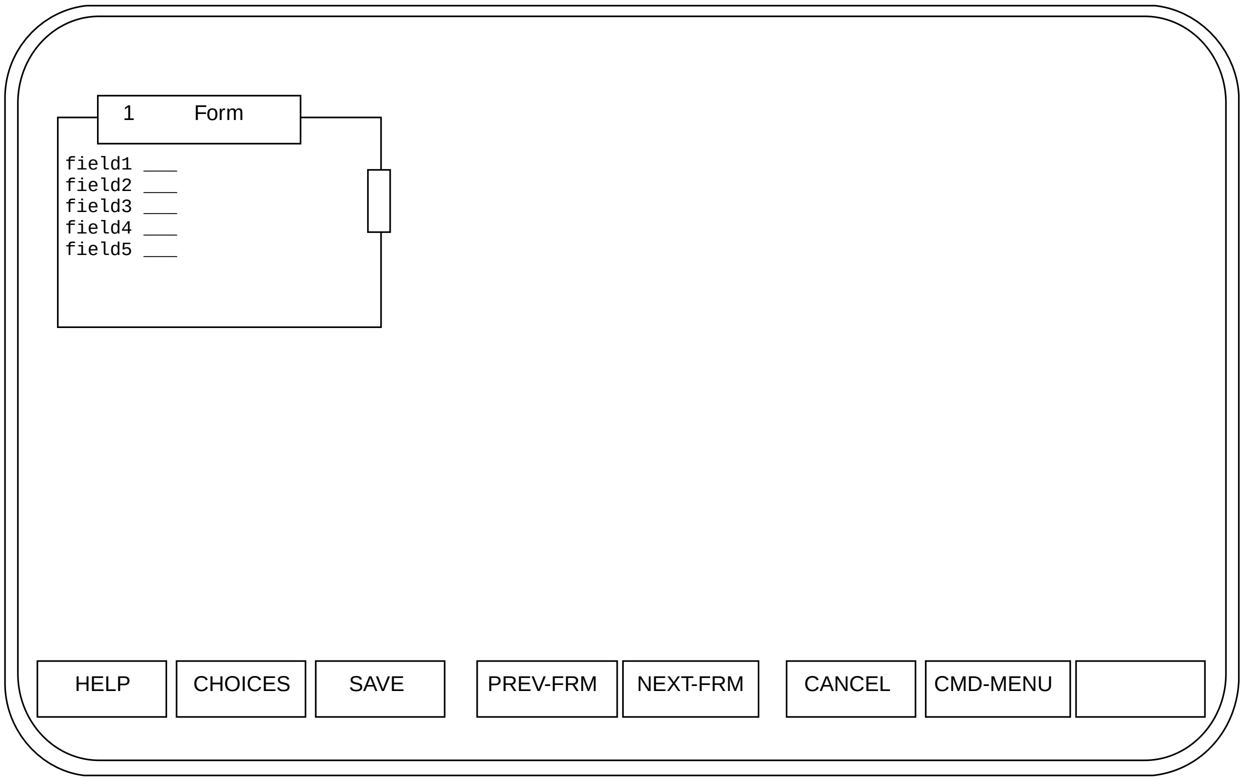
- labels without corresponding fields (for precisely formatted descriptive text within a form)
- fields without any labels

Some applications have made use of these capabilities and will not be broken.

A few simple examples will help. A 5-field form can be defined with only 5 field descriptors, instead of the 30 previously required:

```
autolayout=true
name=field1
name=field2
name=field3
name=field4
name=field5
```

These fields would appear in the form as:

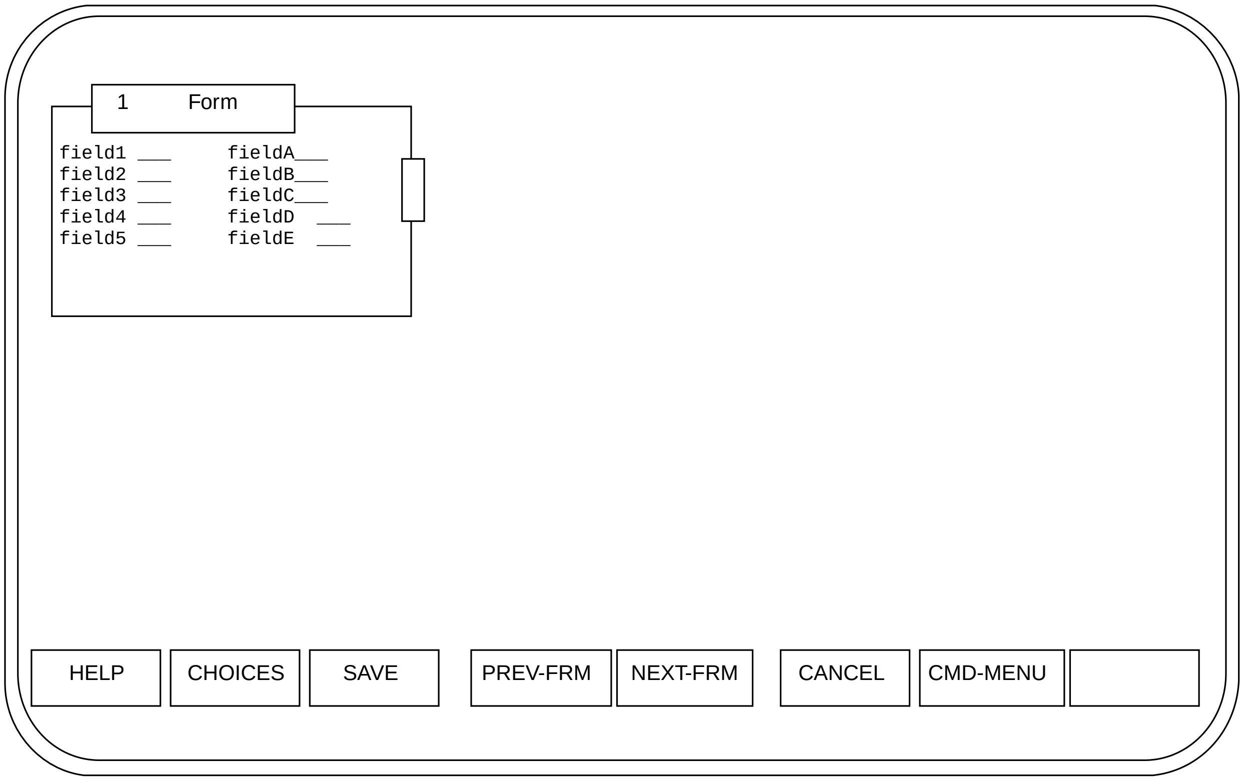


```
field1 ___
field2 ___
field3 ___
field4 ___
field5 ___
```

A simple form with 2 columns and some other variations could be defined with:

```
autolayout=true
name=field1
name=field2
name=field3
name=field4
name=field5
name=fieldA
nrow=0
ncol=14
columns=2
name=fieldB
name=fieldC
name=fieldD
fcol=25
name=fieldE
```

These fields would appear in the form as:



```
field1 ___     fieldA___
field2 ___     fieldB___
field3 ___     fieldC___
field4 ___     fieldD  ___
field5 ___     fieldE  ___
```

Figure 3-26 in the next section shows how defaults would work for a less rigid arrangement of fields and labels.

## Example Form Definition Files

Many of the frame descriptors for forms and menus have the same names and do the same things. Since they were discussed in the examples of menu definition files, they won't be covered again here. Some frame descriptors in forms are different from those in menus, however.

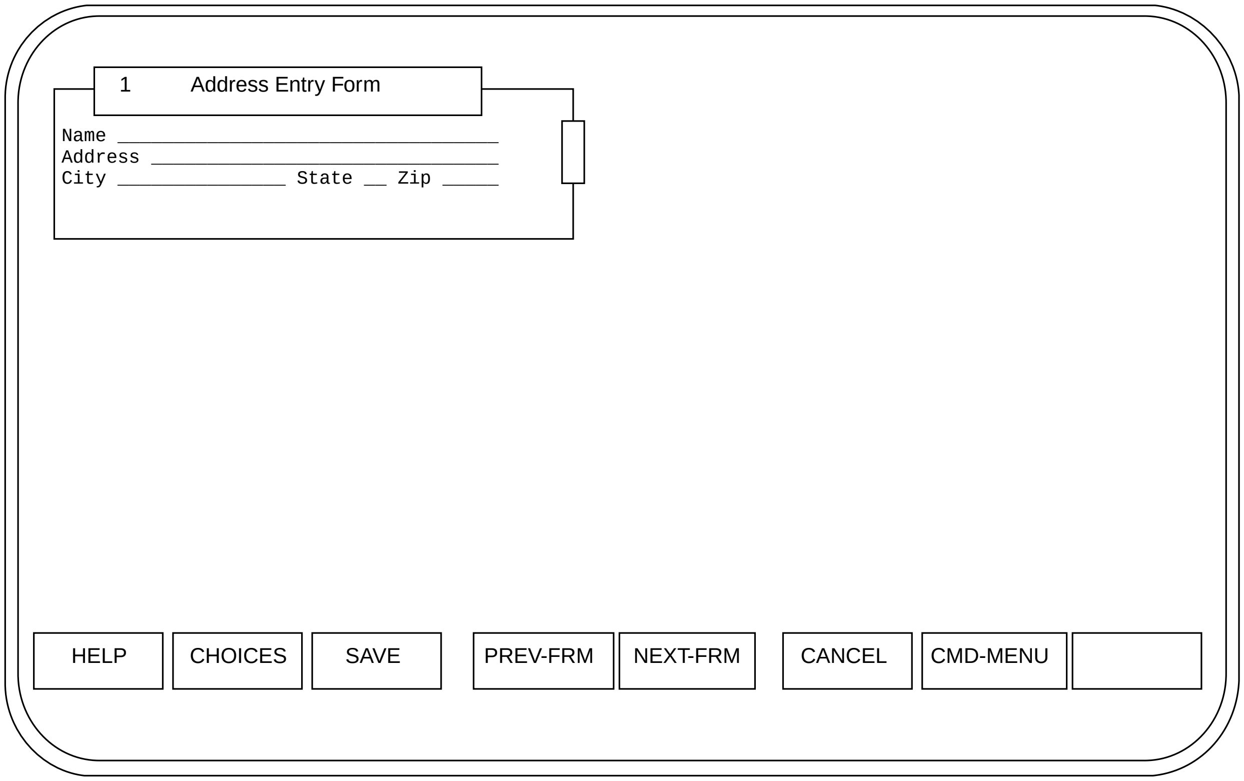
### Saving User Input to a Form

You can use the frame descriptor `done` to take information entered in a form by a user and save it in a file. The address entered by the user in the following form is to be written in a file named **`Addr.file`**. Figure 3-24 shows how the form definition file would look if you coded the field descriptors explicitly with values; Figure 3-25 shows the form itself; Figure 3-26 shows how the definition file would look if you took advantage of the defaults.

```
form=Address Entry Form
done=`echo Name=$F1 >> Addr.file;\
echo Address=$F2 >> Addr.file;\
echo City=$F3 >> Addr.file;\
echo State=$F4 >> Addr.file;\
echo Zip=$F5 >> Addr.file`update

name=Name
nrow=0
ncol=0
frow=0
fcol=5
rows=1
columns=34

name=Address
nrow=1
ncol=0
frow=1
fcol=7
rows=1
columns=31

name=City
nrow=2
ncol=0
frow=2
fcol=5
rows=1
columns=15

name=State
nrow=2
ncol=21
frow=2
fcol=27
rows=1
columns=2

name=Zip
nrow=2
ncol=30
frow=2
fcol=34
rows=1
columns=5
```

**Figure 3-24.  Form.addr: Defaults Not Used**

This frame definition file creates the following form:



```
Name __________________________________
Address _______________________________
City _______________ State __ Zip _____
```

**Figure 3-25. Form.addr: Screen Output**

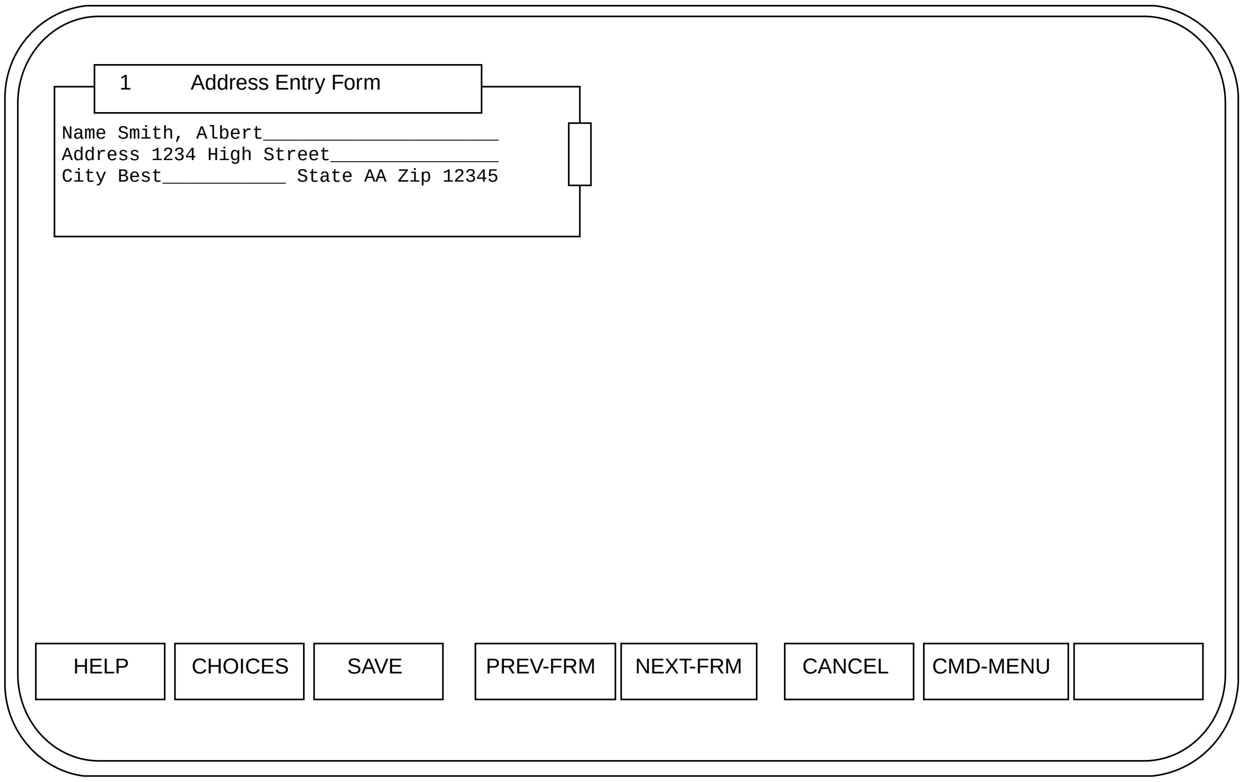
So does the next form definition file, which takes advantage of the default values for field descriptors:

```
form=Address Entry Form
autolayout=true
done=`echo Name=$F1 >> Addr.file;\
echo Address=$F2 >> Addr.file;\
echo City=$F3 >> Addr.file;\
echo State=$F4 >> Addr.file;\
echo Zip=$F5 >> Addr.file`update

name=Name
columns=34

name=Address
columns=31

name=City
fcol=5
columns=15

name=State
nrow=2
ncol=21
columns=2

name=Zip
nrow=2
ncol=30
columns=5
```

**Figure 3-26. Form.addr: Defaults Used**

As the example suggests, you can save yourself considerable effort by using the default values for field descriptors, if you have coded the `autolayout` descriptor as TRUE. Note that `fcol` must be coded for `City` because, by default, FMLI takes the greater of `fcol` for the previous field (7) or `1+`*current_ncol*+*lengthOf Label* (5). That is, you want the input area for `City` to be separated from its label by one space, not three. `nrow` must be coded for `State` because, by default, FMLI increments its value in the previous field (2) by the number of rows in the previous field (1). That is, you want `State` to appear in the same row as `City`, not the fourth row. The same thing holds for the `nrow` descriptor in the `Zip` field. Finally, `ncol` must be coded for the `State` and `Zip` fields because, by default, FMLI uses its value in the previous field. That is, you do not want different fields in the same row to start in the same column.

This form can be used in an application where addresses have to be entered into the system. If a user fills in this form as follows:



```
Name Smith, Albert_____________________
Address 1234 High Street_______________
City Best___________ State AA Zip 12345
```

**Figure 3-27. Form.addr: Screen Output after Being Filled Out by a User**

when the user presses SAVE (or CTRL-f 3), the done descriptor is evaluated and the following information is written into the **Addr.file** file:

```
Name=Smith, Albert
Address=1234 High Street
City=Best
State=AA
Zip=12345
```

**Figure 3-28. Addr.file: Contents after User Saves the Form**

If **Addr.file** does not exist it is created. If it already exists, the information above is appended to it.

Note that the done descriptor in forms is of type command, and thus must evaluate to an FMLI command. In this example, done evaluates to the FMLI command **update.** After the user input is saved in the file **Addr.file,** the **update** command causes the form to be updated to its default values (a blank form), the cursor is positioned on the first field, and the user can begin to enter a new address record in the Address Entry Form.

## Validating a Form Field

Let us add some more functionality to the form. Assume that in this form it is valid to enter addresses of only three states: New York, New Jersey, and Connecticut. To do this, three more field descriptors are added to the `State` field in the frame definition file:

```
.
.
.

name=State
nrow=2
ncol=21
columns=2
rmenu={ CT NJ NY }
menuonly=true
invalidmsg=Valid state codes are: CT, NJ, NY
.
.
.
```

**Figure 3-29. Form.3choices: An Example of Field Validation Using the menuonly Descriptor**

This form definition file will create the same form as above. However, the `rmenu` descriptor is used to create a choices menu, and the `menuonly` descriptor defines only the choices in the choices menu to be valid field values. If the user enters an invalid state code in the `State` field and presses RETURN or SAVE, the message defined by the `invalidmsg` descriptor appears on the message line and the `done` descriptor will not be evaluated.

When the user presses CHOICES the first valid state code, `CT`, will be displayed in the `State` field (see the description of `toggle` in Chapter 4 for a discussion of how to change this default behavior). Each press of CHOICES displays the next valid state code (wrapping to the first choice after the last one is displayed).

Now we will add some more state codes to the list of states in our `rmenu` descriptor and we will change the `invalidmsg` descriptor to reflect the changes:

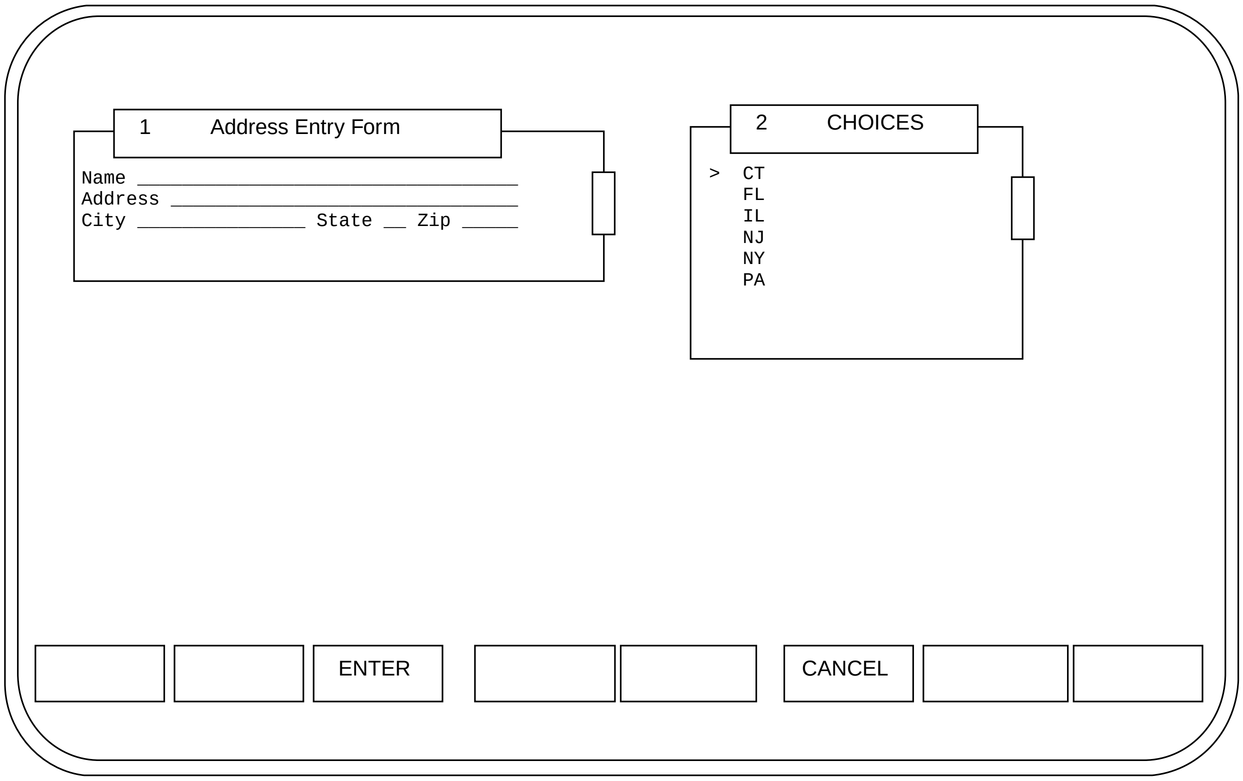
```
.
.
.
name=State
nrow=2
ncol=21
columns=2
rmenu={ CT IL NJ NY PA FL }
menuonly=true
invalidmsg=(Press CHOICES to see valid state codes)
.
.
.
```

**Figure 3-30. Form.6choices: An Example of a Choices Menu**

This frame definition file creates the same form as before. But now if the user enters a wrong state code, the following message will be displayed at the bottom of the screen:

```
(Press CHOICES to see valid state codes)
```

To find the valid state codes, the user can again press CHOICES. This time the valid choices are displayed in a pop-up choices menu. This behavior occurs by default when there are more than three choices in the `rmenu` list:

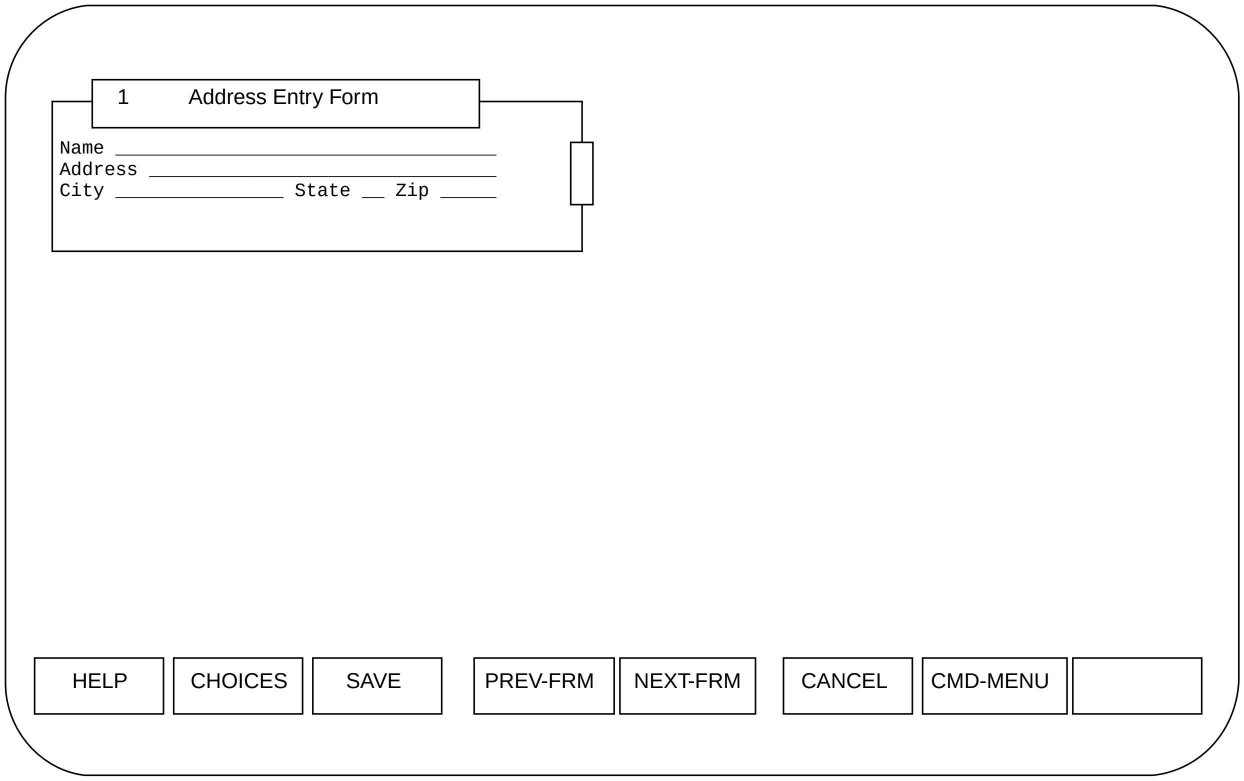


```
Name ___________________________________
Address ________________________________
City _______________ State __ Zip _____
```

```
>  CT
   FL
   IL
   NJ
   NY
   PA
```

**Figure 3-31. Form.6choices: Screen Output**

When the user selects a code from the choices menu, the code is placed in the `State` field and the choices menu is closed, as shown in Figure 3-32:



```
1        Address Entry Form
Name ____________________________________
Address _________________________________
City ________________ State __ Zip _____
```

**Figure 3-32. Form.6choices: Screen Output after User Selects an Item from the Choices Menu**

## Example of Validating a Field Value with the valid Descriptor

Let us add two more descriptors, `valid` and `invalidmsg`, to the `Zip` field definition, as follows:

```
.
.
.
name=Zip
nrow=2
ncol=30
columns=5
valid=`regex -v “$F5” '[0-9]{5}'`
invalidmsg=Zip code is numeric. Five digits only.
.
.
.
```

**Figure 3-33. Form.valid: An Example of Field Validation Using the valid Descriptor**

The `valid` descriptor is defined to execute **`regex`** to validate the field. If a pattern is matched by **`regex`**, **`regex`** will write the corresponding template to `stdout`. The **`regex`** utility will also return the value TRUE which, for FMLI built-in utilities, is analogous to a UNIX system command that exits with status 0. If no pattern is matched, **`regex`** will not write to `stdout` and will return FALSE. For FMLI built-in utilities, FALSE is equivalent to a UNIX system command that exits with a non-zero exit status. For example, the field validation shown in Figure 3-33 (that is, the `valid` descriptor definition) uses the **`-v`** option to **`regex`** to specify that the argument that follows (rather than `stdin`) should be used as input. Note that this **`regex`** statement contains a single pattern without a template. In **`regex`**, a template is optional if only one pattern exists. The last pattern in a series of pattern/template pairs is also optional.

This use of **`regex`** will return TRUE if the current value of field 5 consists entirely of integers and will return FALSE otherwise. Since no template exists, **`regex`** will not write to `stdout`. If the field is not numeric and not all five digits are entered, the following message defined by the `invalidmsg` descriptor is displayed:

```
Zip code is numeric. Five digits only.
```

## Text Frames

A text frame definition file begins with a single set of frame descriptors (ones that define attributes of the whole text frame), and it can end with one or more optional sets of SLK descriptors that define the screen-labeled function keys (one set per SLK) that will be displayed when the text frame is the current frame in the user's work area.

Some of the attributes of a text frame that you can define are the following:

- the title of the text frame
- a non-scrolling header for the text frame
- the text to be displayed
- whether users can modify the text
- the position of the text frame in the work area
- the longevity of the text frame
- new labels and functions for the SLKs

The descriptors in a text frame definition file must follow this order:

*frame_descriptor_1*
.
.
.
*frame_descriptor_n*
[*SLK-n_descriptor_1*
.
.

```
.
```
*SLK-n_descriptor_n*
```
...]
```

**NOTE**

Out-of-order descriptors will be ignored if this order—frame, then SLKs—is not followed.

## Text Frame Descriptors

The set of frame descriptors can be any valid frame descriptors for text frames, in any order. (However, a text frame usually starts with the line `title=`*title*. The default value for `title`, if you do not define it, is `Text`.) If a descriptor is defined more than once in the set, the last one is used.

| | |
|---|---|
| `altslks` | The `altslks` descriptor defines whether SLKs 9 through 16 are displayed when the frame is initially opened. If `altslks` evaluates to TRUE, SLKs 9 through 16 will be displayed. The default, if this descriptor is not defined, is FALSE, which causes SLKs 1 through 8 to be displayed. |
| `begrow`, `begcol` | The `begrow` and `begcol` descriptors define the original position of the top left corner of the text frame in the user's work area. (`begrow=0` and `begcol=0` evaluates to the upper left corner of the work area.) These descriptors accept values of type position: |

| | |
|---|---|
| `center` | the text frame will be centered in the work area |
| `current` | the text frame overlaps the current frame's position (valid for `begrow` only) |
| `distinct` | the text frame will not overlap the current frame (if possible) (valid for `begrow` only) |
| `any` | FMLI chooses a position with least amount of total overlap |
| *integer* | the text frame will be positioned in an absolute position, defined by *integer*. Defining `begrow` and `begcol` to be integer values causes the frame to appear in the given position. |

If either `begrow` or `begcol` evaluates to `center`, then the other can only be an integer value or `center`. Any other value is ignored and the descriptor defaults to `center`.

If neither is `center`, then the value of `begrow` determines the legal values of `begcol`: if `begrow` is `current`, `distinct`, `any`, or an invalid value, then `begcol` defaults to `any`. If `begrow` is a valid integer, `begcol` can be a valid integer; if `begcol` is an invalid integer in this case, it defaults to `any`.

| | |
|---|---|
| `close` | The `close` descriptor is evaluated when the text frame is closed and when the user exits from the FMLI application. The `close` descriptor is of type null, which means its only purpose is to obtain the side effects of a backquoted expressions coded in its definition. |
| `columns` | The `columns` descriptor defines the width of a text frame. It must evaluate to an integer value greater than `0` and less than `DISPLAYW-4`. The `columns` descriptor defaults to `30`. |
| `done` | The `done` descriptor is evaluated when the user executes the **`cancel`** command. If `done` is not defined, it defaults to the FMLI command **`close`**. |
| `edit` | If this descriptor evaluates to TRUE, then the user can modify the text. Otherwise, the text is read-only. The default for this descriptor is FALSE.<br><br>If the user modifies the text in the displayed text frame, this does not modify the frame definition file. |
| `framemsg` | The `framemsg` descriptor displays its value on the message line for as long as the frame is current. It can be temporarily replaced by a message displayed when:<br>• a message is generated by the **`message`** built-in utility with the **`-t`** option<br>• an FMLI error message is generated<br><br>It can be replaced for as long as the frame is current by a message generated by the **`message`** built-in utility with the **`-f`** option. (See the **`message(1F)`** manual page.) |
| `header` | The `header` descriptor defines information that will remain permanently displayed at the top of a text frame. For example, if a text frame contains a long table of information, `header` can be used to define column headings that will remain displayed below the title of the text frame while the user pages or scrolls through the rest of the table. The text defined in `header` can include embedded newline characters, and will be left justified in the frame. The header text will not occupy all rows of the text frame: at least two lines will remain available for display of the text. |
| `help` | The `help` descriptor specifies what will happen when the user requests help while in this text frame. Since this descriptor is evaluated when the user requests help, the specification of what help is displayed can be determined through parameters that are set interactively. |
| `init` | If the `init` descriptor evaluates to FALSE, the frame will not be displayed. If `init` evaluates to FALSE on an update, the frame is closed, unless it is an initial frame. |
| `interrupt` | The Boolean descriptor `interrupt` defines whether an executable that is coded in the `done` descriptor can be interrupted by |

users (FALSE means not interruptible, TRUE means interruptible). It is subject to an inheritance hierarchy: if not defined anywhere in your application, the default value FALSE applies throughout. If explicitly defined at any inheritance level, then executables in `action` and `done` descriptors at or above that inheritance level will inherit that defined value. (See "Interrupt Signal Handling" on page 2-39 for complete information.)

If defined among the frame descriptors in a text frame definition file, that value of `interrupt` is inherited by the `action` descriptor in all sets of SLK descriptors in the text frame, unless it is redefined for a specific SLK.

`lifetime`

The `lifetime` descriptor defines when the text frame will be closed (that is, removed from the work area). It is evaluated whenever the text frame is opened, closed, made current, or made noncurrent. The acceptable values are:

| | |
|---|---|
| `shortterm` | the text frame closes whenever the user navigates to another frame or when the command line is accessed (the user presses CTRL-j or CTRL-f c) |
| `longterm` | the text frame closes when the user issues a **`cleanup`** or **`close`** command |
| `permanent` | the text frame closes whenever the user issues a **`close`** command |
| `immortal` | the text frame closes only when the user exits from the application |

The `lifetime` descriptor is ignored in text frame definition files that are given as arguments when **`fmli`** is invoked: such text frames have a lifetime of `immortal`. See "Other Useful Examples" on page 3-61 for an example of how to use this descriptor to close a frame when another frame is opened or updated.

`oninterrupt`

The command descriptor `oninterrupt` defines what will happen when an interrupt signal is received. If `interrupt` is not coded anywhere in your application, or if it evaluates to FALSE, `oninterrupt` is ignored.

`oninterrupt` is subject to an inheritance hierarchy: if not defined anywhere in your application, the default value `` `message Operation interrupted!` nop `` applies throughout. If explicitly defined at any inheritance level, then executables in `action` and `done` descriptors at or above that inheritance level will inherit that defined value. (See "Interrupt Signal Handling" on page 2-39 for complete information.)

If defined with the frame descriptors in a text frame definition file, that value of `oninterrupt` is inherited by the `action` descriptor in all sets of SLK descriptors, unless redefined for a specific SLK.

| | |
|---|---|
| `reread` | If `reread` is not defined, it defaults to FALSE. If `reread` evaluates to TRUE, the text frame will be periodically updated by rereading its description file when the **`checkworld`** command is executed. **`checkworld`** is executed when a `SIGALRM` alarm occurs (every `$MAILCHECK` seconds). Other times **`checkworld`** is executed include when a frame is opened, closed, or navigated to. (See **`checkworld`** in "Built-in Variables" on page 2-7.) When **`checkworld`** occurs, all frames whose `reread` descriptor evaluates to TRUE will be updated. (However, the `title` descriptor is not reread.) Execution of **`checkworld`** may cause the message line to clear. |
| `rows` | The `rows` descriptor defines the desired number of rows long a text frame will be. It must evaluate to an integer value greater than 0 and less than `DISPLAYH-2`. It defaults to the lesser of 10 or the number of rows needed to display the complete text. |
| `text` | The `text` descriptor defines the text you want to display. It may contain embedded newlines, as long as the value of the entire descriptor is enclosed in quotes. Two special characters are also available for requesting tabs and newlines in the displayed text: |

| | |
|---|---|
| `\n` | insert a newline in this position |
| `\t` | insert a tab in this position |

The alternate character set characters can be coded as well; see "Using the Alternate Character Set" on page 2-42.

| | |
|---|---|
| `title` | The `title` descriptor defines the title of the text frame that will appear in the title bar. It will be truncated if it is longer than `DISPLAYW-6`. If not defined, it defaults to the string `Text`. |
| `wrap` | If this descriptor is set to anything except FALSE, the text will be wrapped to fit the available space when it is read in. If `wrap` evaluates to TRUE, word boundaries are respected at newlines. If not defined `wrap` defaults to TRUE. Newlines in your text are always preserved. |

**NOTE**

Screen labels and actions for function keys can be defined in a text description file as well as in an initialization file. Each set of screen-labeled function-key descriptors must include the `name` and `button` descriptors, and the `name` descriptor must be first in each set. If a descriptor appears more than once in a set, the last one is used.

See Chapter 4 for a discussion of how to use SLK descriptors.

## The textframe Command

The two mechanisms available to present information to the user are the message line and text frames. Messages are limited to 1 line in length; thus, longer information must be put into a text frame. However, writing a separate text frame definition file can be inconvenient if the text frame is very short and needs none of the special capabilities of text frame definition files. Some applications may need hundreds of such short text frames. A shortcut mechanism, the **`textframe`** command, allows applications to be more compact.

There are several advantages to the **`textframe`** command:

- separate text frame definition files do not have to be defined or read-in by your application.
- messages are easier to maintain, since they are coded in the same file as the associated action
- application developers may be more inclined to write useful information more frequently if it can be done in the same file in which it is used

The **`textframe`** command is not a built-in utility; it cannot be used in back-quoted expressions. If it is coded incorrectly, brief error messages will be issued on the message line and the portion coded incorrectly will be ignored. FMLI will also ignore options and arguments it cannot recognize and will use appropriate defaults in those cases. The syntax of this command is

`textframe [`*options*`] “`*text*`”`

The *text* argument, corresponding to the `text` descriptor in text frames, is the only argument to this command. For example, an `action` descriptor could be coded as

```
action=textframe “This is text for a 1-line text frame”
```

If no *text* argument is coded, an empty text frame will appear, just as would a text frame from a definition file whose `text` descriptor is not coded. The *text* argument may contain embedded newlines as well as the notation `\n` to request newlines in the text. The *text* argument must be enclosed in quotes if it includes embedded whitespace or special characters. Thus

```
action=textframe “line1
line2
line3”
```

as well as

```
action=textframe “line1\nline2\nline3”
```

are both allowed (and are equivalent).

### Options for the textframe Command

A text frame opened with the **`textframe`** command is a text frame in all respects, just like one defined in a frame definition file. However, only a subset of the text frame descriptors can be specified via options to this command. The options to the **`textframe`** command, and the text frame descriptors to which they correspond, are:

| Option | Descriptor |
|---|---|
| **`-t`** *title* | `title` |
| **`-l`** *lifetime* | `lifetime` |
| **`-f`** *text* | `framemsg` |
| **`-r`** *integer* | `rows` |
| **`-c`** *integer* | `columns` |
| **`-p`** *position* | `begrow` |
| **`-a`** | `altslks` |

The defaults and valid arguments for these options are the same as for the corresponding descriptors in the text frame, except:

- the **`-l`** *lifetime* option defaults to `shortterm` because the expected use of **`textframe`** frames is for short-term information
- the only valid arguments to the **`-p`** option are `center` and `current`

These options can take arguments, and the arguments must be enclosed in quotes if they contain whitespace. For example:

```
action=textframe -t “Frame Title” “line1\nline2\nline3”
```

Other notes on the behavior of these options and arguments:

- A null string argument to the **`-f`** option (**`-f`** “”) can be used to temporarily turn off a message of permanent duration.
- Use of the **`-a`** option corresponds to coding `altslks=TRUE` in a text frame definition file; this assumes that you have defined at least one of SLKs 9-16 in the FMLI initialization file.
- The alternate character set feature of text frames will work with the text given as an argument to the **`textframe`** command.

## Example Text Frame Definition Files

The following examples will show you how to write a text frame definition file. We'll begin by looking at a simple use of the frame descriptors for text frames, and build from there.

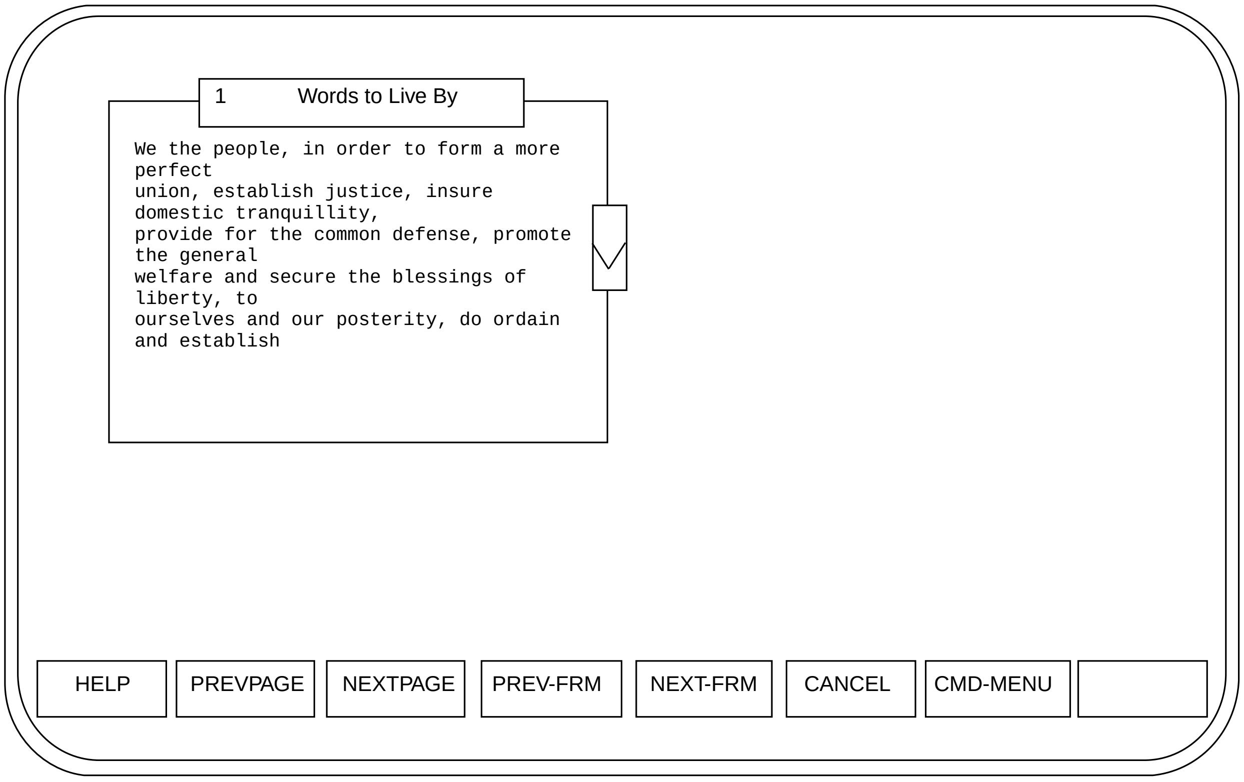
## Defining Attributes of Text Frames

Here is a simple description file for a text frame:

```
title="Words to Live By"
columns=40
lifetime=longterm
wrap=true
text="We the people, in order to form a more perfect
union, establish justice, insure domestic tranquillity,
provide for the common defense, promote the general
welfare and secure the blessings of liberty, to
ourselves and our posterity, do ordain and establish
this constitution for the United States of America."
```

**Figure 3-34. Text.USA: An Example of a Text Frame**

The `lifetime` descriptor defines this text frame to remain on display until the user issues a **`cleanup`** or **`close`** command. The `wrap` descriptor defines word-wrapping to occur: that is, a word that will not fit entirely on the current line will be displayed in full on the next line. The frame will look like this:



```
1          Words to Live By

We the people, in order to form a more
perfect
union, establish justice, insure
domestic tranquillity,
provide for the common defense, promote
the general
welfare and secure the blessings of
liberty, to
ourselves and our posterity, do ordain
and establish
```

**Figure 3-35. Text.USA: Screen Output**

Notice that even though the text is wrapped at 40 columns, the original newline characters are preserved.

## Defining a Text Frame with readfile and longline

A more interesting way to define a text frame is to use the built-in utilities **`readfile`** and **`longline`**:

```
title=“This is a Text Frame”
lifetime=longterm
text=“`readfile $ARG1`”
columns=`longline`
```

**Figure 3-36. Text.readfile: An Example of Using readfile and longline in a Text Frame**

**`Text.readfile`** illustrates the use of arguments that may be passed to menu, text, or form frames. You don't have to define a separate text frame definition file for each file that is to be displayed. Instead, pass `$ARG1` to the text frame when you open it.

For example, if **`Text.readfile`** were opened by a line in a menu that looked like this:

```
action=open $DEF_FILES/Text.readfile help1
```

`$ARG1` would evaluate to `help1`, that file would be read by the built-in utility **`read-file`**, and all of the text in `help1` would become the value of the `text` descriptor, which would then be displayed in a text frame as wide as the longest line of text in the file **`help1`**. For more on how this happens, see “Variables” on page 2-6, and the **`readfile (1F)`** manual page.

## Using Text Frame Headers and Terminal Attributes

Text frame headers, defined by the `header` descriptor, are useful when you want to permanently display some information, perhaps a warning, or headings for columns of information, while the text of the text frame can be scrolled through by the user. The example below in Figure 3-37 also illustrates the use of the terminal display attribute for underlining.

```
title=“Department Directory”
columns=30
rows=5
lifetime=longterm
wrap=true
header=“\+ul  Name            Phone Number  \-ul”
text=“Adams, Jane         663-1234
Brown, Tom          687-3443
Deering, Julia      779-6801
Fitzworth, Leslie   299-7775
Flemming, Eric      344-2289
Shultz, Michael     794-1100
Walinsky, Richard   555-8827
Younger, Helen      865-0023”
```

**Figure 3-37.  Text.header: An Example of Text Frame Headers**

This frame definition file results in the following display:

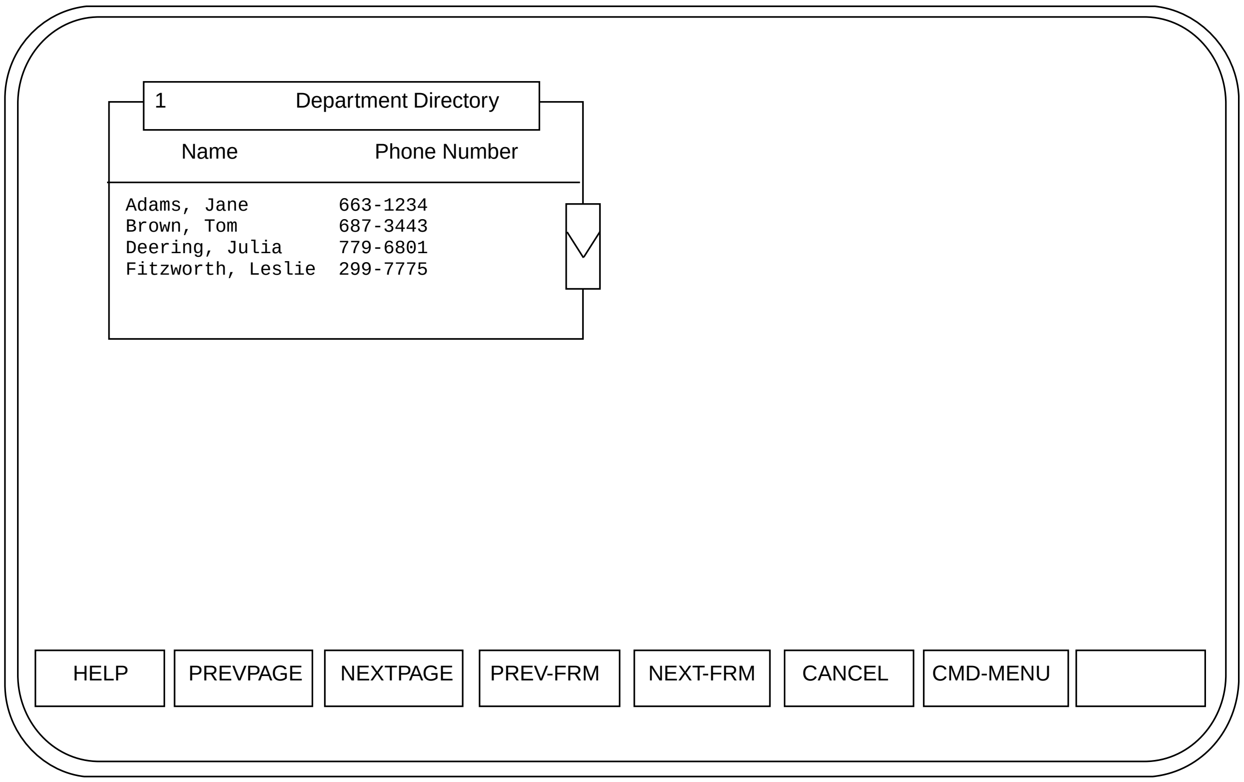

```
1              Department Directory
     Name              Phone Number

Adams, Jane        663-1234
Brown, Tom         687-3443
Deering, Julia     779-6801
Fitzworth, Leslie  299-7775
```

**Figure 3-38.  Text.header: Screen Output**

The terminal attribute of underlining has been turned on for the text header to set it apart from the list of department members. Since the frame is defined to be five rows long, and the header uses one of those rows, the text is displayed in four rows, and the scroll bar indicates that there is more text that follows.

## Other Useful Examples

### Defining a Help Frame for Menu Items or Form Fields

In this example let us take a look how the item descriptor `lininfo` and the built-in variable `LININFO` can be used in conjunction with the frame descriptor `help` to define help that is specific to the current menu item.

This example defines a menu titled `TOP MENU` that has three items. The first item does not have any help information. When help is requested while the cursor is on this item, the help text frame for the menu is displayed. The second and third item have help text frames associated with each item. The menu definition file for this menu is:

```
menu=TOP MENU
help=`if [ $LININFO = “” ];
        then echo open Text.gen_help;
        else echo open '$LININFO';
       fi`

name=Item 1
action=nop

name=date
action=`date | message`nop
lininfo=Text.item2

name=exit
action=exit
lininfo=Text.item3
```

**Figure 3-39. Menu.lininfo: An Example of Defining Help with LININFO**

The text frame definition file **`Text.gen_help`** displays information appropriate to the menu as a whole. (Presumably this information is also sufficient for a user to understand how to use the first menu item, `Item 1`.)

```
columns=20
lifetime=shortterm
title=“Help on TOP MENU”
text=“This menu demonstrates the lininfo descriptor.
The first item does not use the lininfo descriptor.”
```

**Figure 3-40. Text.gen_help: An Example of a Help Text Frame**

The text frame definition files **`Text.item2`** and **`Text.item3`** display appropriate help information for the second and third items on the menu.

```
columns=20
lifetime=shortterm
title=“Help on date”
text=“The selection of this item will display the
current date and time on the message line”
```

**Figure 3-41. Text.item2: An Example of a Help Text Frame**

```
columns=20
lifetime=shortterm
title=“Help on exit”
text=“This item will let you get out of the application”
```

**Figure 3-42. Text.item3: An Example of a Help Text Frame**

When this application is run, the user sees the following menu:

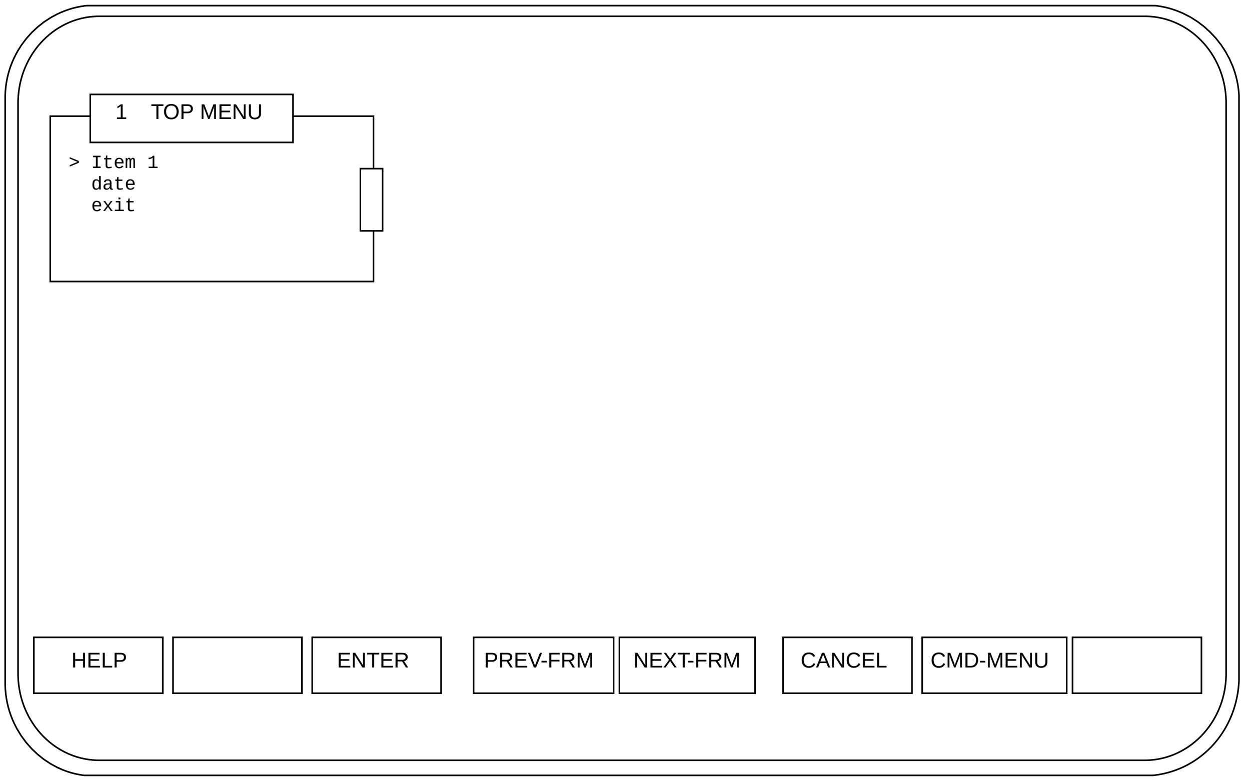
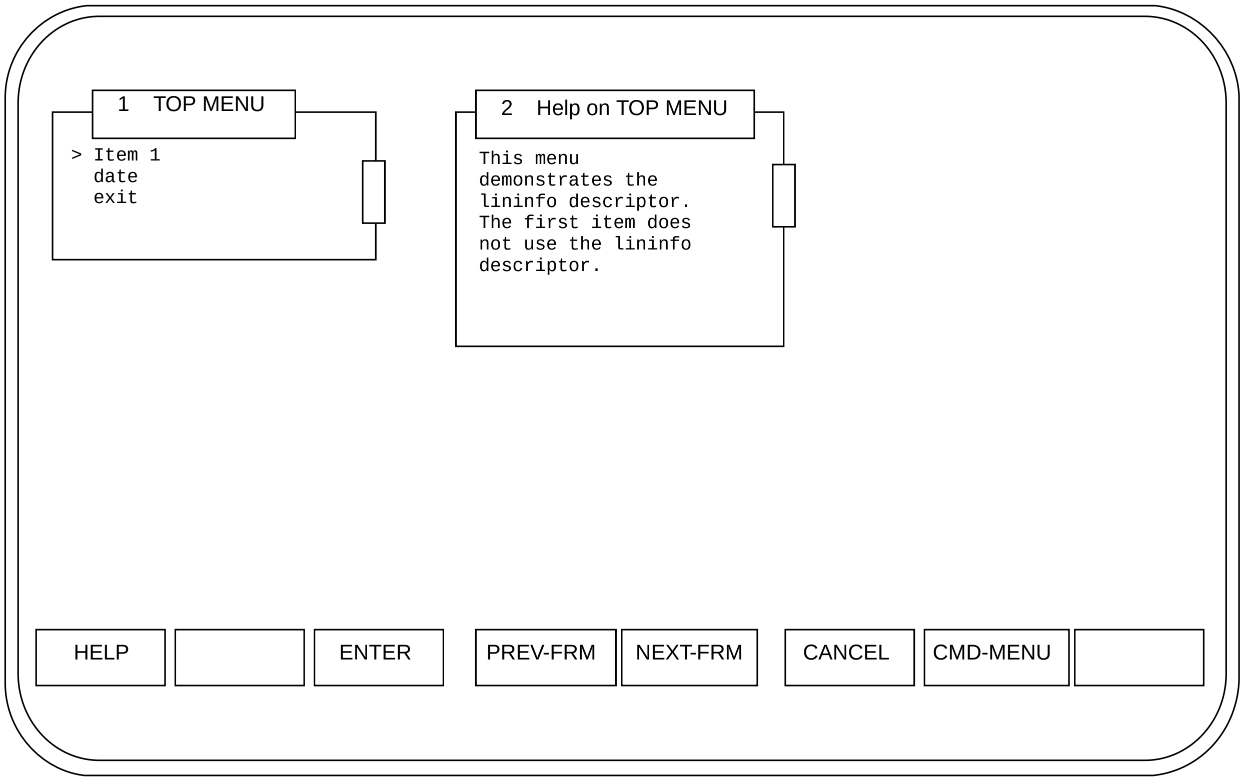


**Figure 3-43. Menu.lininfo: Screen Output**

If the user asks for help, either by the appropriate function key, or by the **`help`** command (from the command line or the `Command Menu`), the appropriate text frame will be displayed. The `lifetime` descriptor, defined in each of the help text frames to be `shortterm`, ensures that whenever the user navigates away from the help text frame, it will be removed from the work area, thus reducing screen clutter:

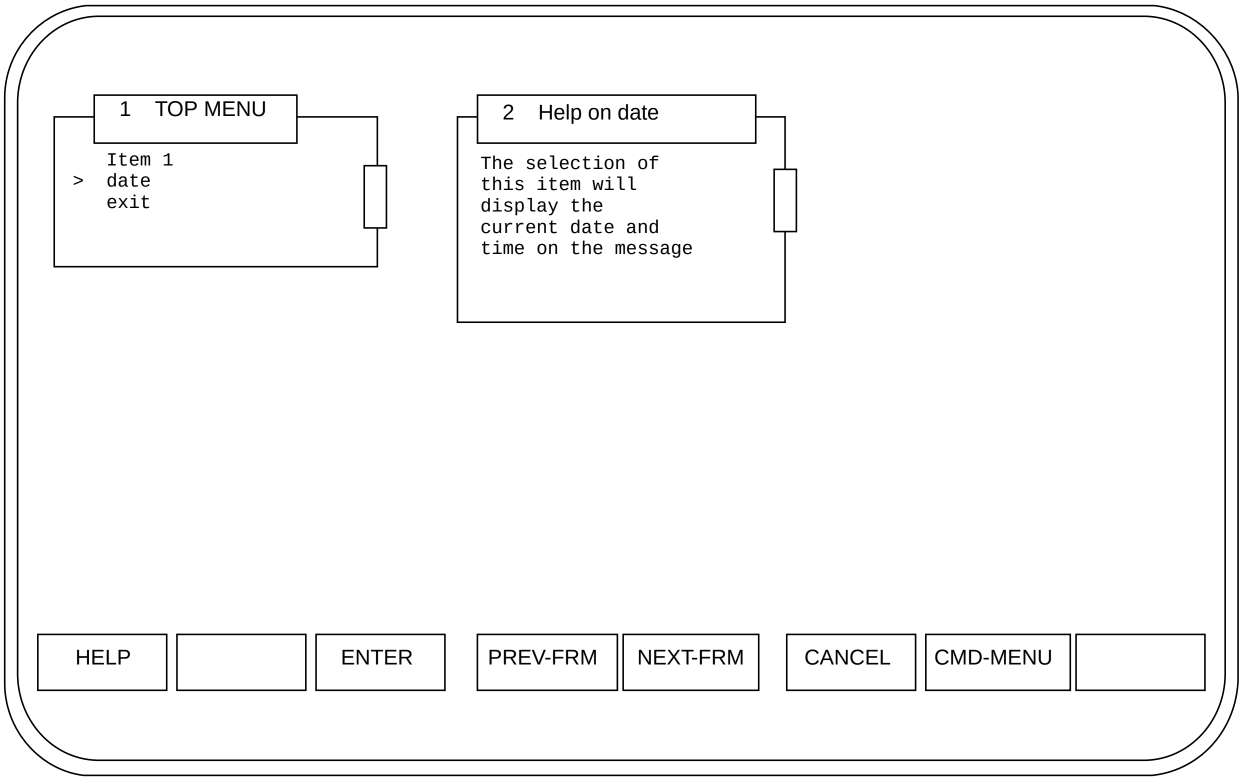


```
> Item 1
  date
  exit
```

```
This menu
demonstrates the
lininfo descriptor.
The first item does
not use the lininfo
descriptor.
```

**Figure 3-44. Menu.lininfo: Screen Output after Requesting Help on Item 1**

If the user navigates to the second menu item and again asks for help, the following text frame will be displayed:



```
   Item 1
>  date
   exit
```

```
The selection of
this item will
display the
current date and
time on the message
```

**Figure 3-45. Menu.lininfo: Screen Output after Requesting Help on Item 2**

The text defined in **`Text.item3`** is displayed in the same manner when `HELP` is pressed while the cursor is positioned on menu item 3, `exit`. The display of these help frames is controlled by the `lininfo` descriptor and the `LININFO` variable.

## Using the textframe Command as an Alternative

The previous example of defining help text frames using the `LININFO` variable could be coded instead using the **`textframe`** command. This would eliminate the need for 3 text frames.

To do so, you would change the line in **`Menu.lininfo`** that is coded as

```
else echo open '$LININFO';
```

to something like

```
else echo textframe -c20 '$LININFO';
```

Then the corresponding values for `LININFO` later in that file would be coded with the text values from the `text` descriptors in the respective text frame definition files in the example. If the different titles for each frame were to be kept as well, then the `LININFO` variable for each item would have to include it:

```
lininfo='-t “frame title” “Text contents”'
```

## Using Co-processing Utilities

Co-processing allows an external process to communicate with the user via a menu, form, or text frame. A co-process does not have direct access to the terminal's screen. It communicates with the FMLI application, which can then post the messages in the frame that contains the co-processing descriptors or take other appropriate actions.

The co-processing feature in FMLI consists of five built-in utilities: **`cocreate`**, **`cosend`**, **`cocheck`**, **`coreceive`**, and **`codestroy`**, which support inter-process communication.

The **`cocreate`** utility is responsible for initializing the co-process and setting up pipes between it and FMLI. The **`codestroy`** utility is responsible for cleaning up when the communication has been completed. The utility **`cosend`** is used to send information to the co-process via the pipe and block (wait) for some response by the co-process. The **`-n`** option to **`cosend`** performs a *no wait* write. This means that **`cosend`** will send information to the co-process but will not block for a response. The **`cocheck`** utility checks the incoming pipe for information. The **`coreceive`** utility performs a “no-wait” read on the pipe. The purpose of these built-in utilities is to provide a flexible means of interaction between FMLI and a co-process; to be responsive to asynchronous activity.

It is important to note that information passed to FMLI from a co-process is treated as text only. FMLI commands (for example, **`open, close, update)`** will not be recognized by FMLI unless they become the value of a descriptor of type command.

To illustrate the use of co-processing, consider a UNIX system program that wishes to “talk” to the user as it executes (an interactive program). The following sample menu displays the item `talk`. When `talk` is selected, the backquoted expression creates the co-process and then opens an “interactive” form, `Form.talk`.

```
menu=“My Menu”
name=“talk”
action=`cocreate -i MYPROC talk` open Form.talk
```

**Figure 3-46. Menu.talk: An Example of Co-processing**

In the form frame definition file **`Form.talk,`** shown in Figure 3-47, the following occurs:

- The `close` descriptor is responsible for destroying the communication.
- The `reread` descriptor checks the pipe and rereads the frame definition file if there is information pending.
- Field 1 is an inactive field, used simply to display text received from the co-process.
- Field 2 is an active field which will get information from the user and send it to the co-process (**`cosend`**). This is done via the `valid` descriptor which is evaluated when a field value changes.

- A SLK, `F8`, is defined to abort the co-process at any time. This is done by forcing a close operation (as usual, the descriptor `close` is evaluated when a frame is closed).

```
form=“Talking ...”
close=`codestroy MYPROC`
reread=`cocheck MYPROC`

name=“”
fcol=0
rows=5
columns=20
inactive
value=“`coreceive MYPROC`”

name=“”
fcol=0
columns=20
valid=`cosend -n MYPROC “$F2”`TRUE

name=abort
button=8
action=`message “Communication stopped ...”`close
```

**Figure 3-47. Form.talk: An Example of Co-processing**

The following code segment illustrates how an interactive co-process (in this case `talk`) may be structured:

```
response=“nothing”
while :
do
echo “I received $response.”
   vsig
   read response
   if [ “$response” = “goodbye” ]
   then
      break
   fi
done
echo “goodbye”
vsig
```

**Figure 3-48. talk: An Example of a Co-process**

The executable **`vsig(1F)`** is used to send a signal telling the interpreter that information is pending. This interrupt causes `reread` to be evaluated. For more information about co-processing, see the **`coproc(1F)`** manual page.

# 4
# Application Level Definition Files

# 4
# Application Level Definition Files

## Introduction

This chapter describes the optional application level files that you can define for your application. The application level definition files define attributes of the application as a whole.

- “The Initialization File” on page 4-1 describes, among other things, how to define an introductory frame (say, a copyright notice), and how to redefine the banner line, the colors to be displayed on color terminals, and the default functions and labels assigned to screen-labeled keys. Examples are given.
- “The Commands File” on page 4-12 describes how to disable or redefine existing FMLI commands, and define new ones. Examples are given.
- “The Alias File” on page 4-14 describes how to define aliases for lengthy path names to files or devices, and how to define search paths (like `$PATH` in the UNIX system shell). Examples are given.
- “fmli Command Syntax” on page 4-15 discusses the syntax of the **`fmli`** command, and explains how to supply names of the initialization file, the commands file, and the alias file as arguments when **`fmli`** is invoked.

## The Initialization File

The initialization file is a file containing descriptors that apply to your application as a whole. If you want to use an initialization file for your application, its name can be supplied as an argument, **`-i`** *initialization_file*, when the **`fmli`** command is invoked. As its name implies, its contents are read when your application is invoked. (It can also be reread using the **`reinit`** command.) In the initialization file you can define the following facets of your application:

- a transient introductory frame, displaying the application name
- a banner, its position, and other elements of the banner line
- the colors of various elements of the FMLI screen
- the behavior of some aspects of your application, such as choices menus, and user access to the UNIX system
- screen-labeled function keys (SLKs) and their layout

A suggested order for initialization file descriptors is the following, although the only order that is enforced is that any sets of screen-labeled function key descriptors must be last in the initialization file.

```
[introductory_frame_descriptor_1
.
.
.
introductory_frame_descriptor_n]

[banner_line_descriptor_1
.
.
.
banner_line_descriptor_n]

[color_attribute_descriptor_1
.
.
.
color_attribute_descriptor_n]

[general_application_descriptor_1
.
.
.
general_application_descriptor_n]

[SLK-n_descriptor_1
.
.
.
SLK-n_descriptor_n
...]
```

Note that all sets of descriptors in an initialization file are optional, as is the initialization file itself.

## Introductory Frame Descriptors

An introductory frame is a frame that is displayed briefly when your application starts, and is then cleared from the screen and replaced by the frame(s) you specify as arguments when **`fmli`** is invoked as the initial frame(s) to open. The introductory frame will be displayed again briefly when the user exits from your application.

The introductory frame is defined with four descriptors normally used to define a text frame. Note, however, that the defaults are different when they are used in an initialization file.

The introductory frame descriptors are described below. Either the `title` or `text` descriptor must be included in the set of introductory frame descriptors.

`columns` The `columns` descriptor defines how many columns wide you want the introductory frame to be. It defaults to the integer value `50` if not defined for an introductory frame.

`rows` The `rows` descriptor defines how many rows high you want the introductory frame to be. It defaults to the integer value `10` if not defined for an introductory frame.

`text` The `text` descriptor defines the text you want to display in the introductory frame. It defaults to NULL if not defined. If neither the `title` descriptor or the `text` descriptor is defined in the initialization file, the introductory frame will not be displayed.

`title` The `title` descriptor defines the title that will appear at the top of the introductory frame. It defaults to NULL if not defined. It will be truncated if it is longer than `DISPLAYW-6`. If neither the `title` descriptor or the `text` descriptor is defined, the introductory frame will not be displayed.

### Example Definition of an Introductory Frame

A definition for an introductory frame is simple, as the following example shows:

```
title="WELCOME TO"
text="My Application
Copyright (c) 1989
My Software, Inc.
All rights reserved."
rows=5
columns=25
```

Backquoted expressions, containing calls to built-in utilities, may also be used, as in this line:

```
text="`readfile DEF_FILES/myintrotext`"
```

which will cause the text frame definition file **`myintrotext`** to be read from the directory whose alias is defined to be **`DEF_FILES`**, and passed to the `text` descriptor as the argument. (See "The Alias File" on page 4-14 for more information about how to define aliases.)

## Banner Line Descriptors

Your application can display a different banner on the banner line. The `banner` descriptor must be included in the set of banner line descriptors.

`bancol` The `bancol` descriptor defines the position of the banner in the banner line. If not defined, this descriptor defaults to `center`. It accepts the following values of type position:

| | |
|---|---|
| `center` | centers the value of `banner` in the banner line |
| *integer* | the banner will begin in the column specified by *integer* |

`banner` The `banner` descriptor defines information that will appear in the banner line on the user's screen while your FMLI application is running. If not defined, it defaults to NULL.

`working` The `working` descriptor defines a string used to notify users that they must wait until FMLI completes an activity. It always appears flush-right on the banner line. If this descriptor is not defined, it defaults to the string `Working`.

**NOTE**

Taking care that other items on the banner do not run into this area is the responsibility of the developer.

### Example Definitions of a Banner Line

The following lines in an initialization file will give you a banner with the program name, the date, and the time the FMLI application was started on the banner line (top line of the screen), starting in the 30th column:

```
banner=“MYPROGRAM - `date`”
bancol=30
```

The `working` icon appears right-justified on the banner line. You can change the working icon, to `BUSY` for example, by defining the `working` descriptor in your initialization file as follows:

```
working=“BUSY”
```

You may also put an application-specific indicator on the banner line by using the built-in utility **`indicator`** (see the **`indicator(1F)`** manual page for complete details on its use).

## Color Attribute Descriptors

The color attribute descriptors allow you to define the colors of various elements of the FMLI screen. The color descriptors can only be defined in the initialization file. They will be ignored in other files.

**`curses(3curses)`** requires that the colors be set in pairs. This means you must set both the foreground and background for a specific element of the screen; otherwise it will default to monochrome. The pair for each color descriptor is indicated in the descriptions that follow.

**NOTE**

If you set the foreground and background to the same color, you will not be able to see the text.

The colors that can be used as values for the color attribute descriptors, for either foreground or background, are the following:

- `black`
- `blue`
- `green`
- `cyan`
- `red`
- `magenta`
- `yellow`
- `white`

You may redefine these colors, or add new ones, with the **`setcolor`** built-in utility (see the **`setcolor(1F)`** manual page for complete details on its use). Of course, if the terminal your application is being run on cannot display color, FMLI automatically defaults to monochrome.

The following descriptors can be used in the initialization file to specify color attributes for the various screen elements. All of these descriptors are of type string and accept the color values listed previously.

If the terminal your application is running on does not support color, these descriptors are ignored. (You can use the built-in variable `HAS_COLORS` to test for color support.)

| | |
|---|---|
| `active_border` | The `active_border` descriptor defines the color of the frame border when a frame is current (border foreground). This will enforce the "solid line" look of the screen border. The background for the active border is defined by `screen`. |
| `active_title_bar` | The `active_title_bar` descriptor defines the color of the title background when a frame is current (background for `active_title_text`). |
| `active_title_text` | The `active_title_text` descriptor defines the color of the title text when a frame is current (foreground for `active_title_bar`). |
| `banner_text` | The `banner_text` descriptor defines the color of all text on the banner line. If this descriptor is not defined in the initialization file, the banner text defaults to white. The background for this text is defined by `screen`. |
| `highlight_bar` | The `highlight_bar` descriptor defines the color of the menu selector bar (background for `highlight_bar_text`). |
| `highlight_bar_text` | The `highlight_bar_text` descriptor defines the color of the menu selector bar text (foreground for `highlight_bar`). |

| | |
|---|---|
| `inactive_border` | The `inactive_border` descriptor defines the color of the frame border when a frame is non-current (border foreground). The background for the inactive border is defined by `screen`. |
| `inactive_title_bar` | The `inactive_title_bar` descriptor defines the color of the title background when a frame is non-current (background for `inactive_title_text`). |
| `inactive_title_text` | The `inactive_title_text` descriptor defines the color of the title text when a frame is non-current (foreground for `inactive_title_bar`). |
| `screen` | The `screen` descriptor defines the color of the screen (screen background) |
| `slk_bar` | The `slk_bar` descriptor defines the color of the screen-labeled function keys (background for `slk_text`). |
| `slk_text` | The `slk_text` descriptor defines the color of the screen-labeled function key text (foreground for `slk_bar`). |
| `window_text` | The `window_text` descriptor defines the color of the text in a frame (text foreground). If this descriptor is not defined in the initialization file, it defaults to white. The background for this text is defined by `screen`. |

### Examples of Defining Color Attributes

The examples below show how to define the color of an area of the screen, and how to use the built-in utility **`setcolor`** example to redefine one of the default color definitions and assign it to a portion of the screen.

### Defining Color for the Banner Line

The color for text on the banner line is controlled by the descriptor `banner_text`. If this descriptor is not set, the default is white text on a background that is the same color as the background for the rest of the screen.

```
banner_text=yellow
```

would make all text on the banner line yellow, and the background would be whatever you set it to for the rest of the screen.

## General Application Descriptors

The following descriptors can be used in the initialization file to define some display and functional characteristics globally for your application.

| | |
|---|---|
| `autolayout` | The `autolayout` descriptor in an initialization file defines whether the reasonable defaults for form field and label position- |

ing available in this release of FMLI will be used. If not coded, it defaults to FALSE. If it evaluates to TRUE, then the reasonable defaults will be used in all forms of the application whose `auto-layout` descriptor is not coded. See “Automatic Layout of Form Fields” on page 3-40 for full information.

This application-level descriptor may be coded TRUE to get the defaults for an entire application, while a particular form's `auto-layout` descriptor may be coded FALSE to be explicitly protected from these defaults. This explicit enabling of the new defaults is necessary to preserve compatibility with older applications; it was possible using the previous defaults of -1 for an application to obtain labels without corresponding fields (to achieve precisely formatted descriptive text) or fields without any labels. Some applications have made use of this capability and will thus not be broken.

`interrupt`

The `interrupt` descriptor in an initialization file defines whether any executable coded in `action` or `done` descriptors in your application can be interrupted by the user. If not coded, it defaults to FALSE. If it evaluates to TRUE, then executables will be interruptible.

If defined in an initialization file, the value of `interrupt` affects executables in all `action` and `done` descriptors—in the SLK section of the initialization file, in all frame definition files, and in the commands file—unless otherwise defined at one of those levels. (See “Interrupt Signal Handling” on page 2-39 for more information.)

`nobang`

The `nobang` descriptor allows you to control user access to the UNIX system shell and UNIX system commands from the command line. If not defined, `nobang` defaults to FALSE (users can access the UNIX system shell).

FMLI allows users to escape to the UNIX system shell from the command line (accessed with `CTRL-j` or `CTRL-f c`) by prefixing an exclamation point (`!`) to the command to be executed in the UNIX system. For example,

```
    -->!pwd
```

But if `nobang` evaluates to TRUE, use of the `!` prefix to commands entered on the command line will be disabled, and a message to that effect is displayed on the message line. In addition, when `nobang` evaluates to TRUE, **open** is also disabled from the command line.

**NOTE**

The `nobang` descriptor does not disable the **`unix-system`** command. See “The Commands File” on page 4-12 later in this chapter for information on disabling access to the UNIX system via the **`unix-system`** command.

`oninterrupt`

The `oninterrupt` descriptor specifies the action to be taken when an interrupt signal is received. If `oninterrupt` is not defined anywhere in your application, it defaults to `` `message Operation interrupted!` nop ``. If `interrupt` is not coded anywhere in your application, `oninterrupt` is ignored.

If defined in an initialization file, the value of `oninterrupt` affects executables in all `action` and `done` descriptors—in the SLK section of the initialization file, in all frame definition files, and in the commands file—unless otherwise defined at one of those lower levels.

(See “Interrupt Signal Handling” on page 2-39 for more information.)

`permanentmsg`

The `permanentmsg` descriptor defines information that will be displayed on the message line until explicitly replaced or removed by another message of permanent duration. (Messages of permanent duration are those defined with `permanentmsg` or with **`message -p`**.)

A message of permanent duration can be temporarily displaced by messages of frame duration or transient duration. When the frame duration or transient duration message expires, the value of the most recent use of `permanentmsg` or **`message -p`** will again be displayed on the message line. (See the **`message(1F)`** manual page for complete information on message durations.)

`slk_layout`

The `slk_layout` descriptor defines the layout of the screen labels for function keys. Two layouts are supported: `4-4` and `3-2-3`. The value `4-4` causes screen labels to be displayed in two groups of four, as follows:

```
F1 F2 F3 F4          F5 F6 F7 F8
```

The value `3-2-3` causes screen labels to be displayed in three groups of three, two, and three, in that order, as follows:

```
F1 F2 F3       F4 F5       F6 F7 F8
```

The default, if this descriptor is not defined, is `3-2-3`.

`toggle`

The `toggle` descriptor defines the way you want valid choices to be displayed when a user presses `CHOICES` in a form field for which you have defined the `rmenu` descriptor. By default, the available choices are shown in the field itself if there are three or fewer choices. If there are more than three choices, a pop-up menu displays the available choices.

You can change this default behavior by defining the `toggle` descriptor in the initialization file. It accepts any of the following values:

| | |
|---|---|
| `always` | When the `CHOICES` function key is pressed, users will always be toggled through the choices in the field itself. |
| *integer* | When the number of choices is greater than *integer*, a pop-up menu will be displayed. When the number of choices is less than or equal to *integer*, toggling will occur. |
| `never` | When the `CHOICES` function key is pressed, the user will never be toggled through choices in the field itself; a pop-up choices menu will always appear. |

**NOTE**

If `toggle` evaluates to less than 1, it defaults to 3.

`use_incorrect_pre4.0_behavior`

The `use_incorrect_pre4.0_behavior` descriptor causes FMLI to re-evaluate variables referenced with the `$` notation until no special characters remain in the expression. If this descriptor is not defined, it defaults to FALSE.

If this descriptor evaluates to TRUE, then the `$` notation behaves in the manner defined for the `$!` notation, and the `$!` notation has no special meaning. (See “Variable Evaluation” on page 2-8 for a complete discussion of the `$` and `$!` notation for variable evaluation.)

**NOTE**

This descriptor, and consequently the ability to make the `$` notation behave like the `$!` notation, will be removed in the next release of FMLI.

## Screen-labeled Function Key Descriptors

Eight labels appear on the last line of the user's screen to indicate the functions currently assigned to the corresponding keyboard function keys `F1` through `F8`. The screen labels are analogous to a set of menu items that are always displayed and from which the user can make a selection at any time by pressing the corresponding function key. If a keyboard does not have function keys, the user can select the function by using one of the alternative keystroke sequences `CTRL-f 1` through `CTRL-f 8`.

**NOTE**

FMLI downloads alternative keystroke sequences into the function keys of some terminals at the user's request. For a discussion, see Appendix B.

FMLI provides two sets of screen labels for function keys. There are eight screen labels in each set. FMLI has preassigned functions to only the first set of eight for each frame type.

**Figure 4-1. Default Screen-labeled Keys**

| Function Key | Menu Frame | Form Frame | Text Frame | Choices Menu | Command Menu |
|---|---|---|---|---|---|
| F1 | help | help | help | | help |
| F2 | mark* | choices | prevpage | | |
| F3 | enter | save | nextpage | enter | |
| F4 | prev-frm | prev-frm | prev-frm | | |
| F5 | next-frm | next-frm | next-frm | | |
| F6 | cancel | cancel | cancel | cancel | cancel |
| F7 | cmd-menu | cmd-menu | cmd-menu | | |
| F8 | chg-keys** | chg-keys** | chg-keys** | chg-keys** | chg-keys** |
| F16 | chg-keys** | chg-keys** | chg-keys** | chg-keys** | chg-keys** |

* Function key F2 is assigned the **mark** command only in multi-select menus. In single-select menus F2 has no default assigned.

** Function keys F8 and F16 will default to **chg-keys** only if any of keys F9 through F15 are defined by the developer.

Function keys F1 through F7 in the first set can be disabled but not redefined. Function keys F8 through F16 may be defined. However, if you define F8 or F16, the user loses the ability to alternate between the two sets of SLKs. If you want to define PREV-FRM, NEXT-FRM, PREVPAGE, or NEXTPAGE, on the second set of function keys, the labels must be spelled exactly as they are on the first set (case is irrelevant).

**NOTE**

When you redefine screen-labeled function keys in the initialization file, your definitions become the defaults. Screen-labeled function keys may also be defined in individual form, menu, and text frame definition files. When they are defined in frame definition files, those definitions override the defaults (either the FMLI-defined defaults, or the defaults you may have defined in the initialization file) while that frame is active.

You can define which set of screen-labeled function keys first appears when a frame is opened by defining the frame level descriptor `altslks` in menu, text, and form definition files. If `altslks` evaluates to TRUE, the screen labels for function keys 9 through 16 will be displayed when the frame is first opened.

The following is a list of the descriptors used to define screen-labeled function keys. The `name` and `button` descriptors must be included in each set of SLK descriptors, and `name` must be first.

**NOTE**

Keep in mind that the screen-labeled function keys must be the last things defined in the initialization file, or in any frame definition file.

| | |
|---|---|
| `action` | The `action` descriptor defines the command to execute when the particular screen-labeled function key is selected. |
| `button` | The `button` descriptor specifies the screen-labeled function key you are defining or disabling. The value of `button` is an integer corresponding to the number of the function key (1 through 16) to which the screen label corresponds. |
| `interrupt` | The `interrupt` descriptor defines whether an executable that is coded in the `action` descriptor can be interrupted by the user. If not coded, `interrupt` defaults to FALSE. If this descriptor evaluates to TRUE, then executables will be interruptible.<br><br>If the `interrupt` descriptor is defined for a SLK in the initialization file, that value is inherited by the SLK unless the SLK is redefined in a frame definition file. Redefining a SLK in a frame definition file completely overrides a definition of it you may have coded in the initialization file. For example, if you define `interrupt` for a particular SLK in the initialization file, but do not include `interrupt` in a redefinition of that SLK in a frame definition file, the SLK will inherit the value of the `interrupt` descriptor defined at the next higher inheritance level (from the frame descriptors if defined there, then from the general descriptors in the initialization file if defined there, then from the FMLI defaults). |
| `name` | The `name` descriptor defines the name that is displayed on the screen label. The value of `name` must be 8 or fewer characters. Defining `name` as a null string (`name=""`) will disable the function key. |
| `oninterrupt` | The `oninterrupt` descriptor specifies the action to be taken when an interrupt signal is received. If it is not defined anywhere in your application, it defaults to `` `message Operation interrupted!`nop ``. It is ignored if `interrupt` is not coded anywhere in your application or if `interrupt` evaluates to FALSE. |

If the `oninterrupt` descriptor is defined for a SLK in the initialization file, that value is current for the SLK, unless the SLK is redefined in a frame definition file. If a SLK is redefined in a frame definition file, all descriptors for that SLK in the initialization file, including the `oninterrupt` descriptor, are ignored. If a SLK definition does not define `oninterrupt`, the SLK inherits the value set for `oninterrupt` from the frame level descriptors, then from the application level section of the initialization file.

### Example Definitions of Screen-labeled Function Keys

The following example shows how to

- disable F7 (labeled CMD-MENU and assigned the FMLI command **`cmd-menu`** by default)
- define F9 (the first SLK in set 2) to execute the **`exit`** command and display the screen label EXIT

in the initialization file:

```
name=“”
button=7

name=“EXIT”
button=9
action=exit
```

## The Commands File

FMLI commands can be added to the Command Menu or disabled. You can also define new, application-specific commands to appear in the Command Menu. This is done in a commands file. If you create a commands file for your application, its name must be supplied as an argument, **`-c`** *commands_file*, when **`fmli`** is invoked. Each set of descriptors is ordered as follows:

```
[command_descriptor_1
.
.
.
command_descriptor_n
...]
```

**NOTE**

There is an absolute maximum of 64 sets of command descriptors in a commands file.

## Command Descriptors

The name and action descriptors must be included in each set of command descriptors, and name must be first in each set.

| | |
|---|---|
| action | The action descriptor defines the operation to perform when the command **name** is selected. |
| help | The help descriptor defines a command to be executed when the user asks for help on name. Since this descriptor is evaluated when the user requests help, the specification of what help is displayed can be determined through parameters that are set interactively. |
| interrupt | The interrupt descriptor defines whether an executable that is coded in the action descriptor in a command definition can be interrupted by the user. If not coded, it defaults to FALSE. If this descriptor evaluates to TRUE, then executables defined for this command's action descriptor will be interruptible.<br><br>If defined among the general descriptors in an initialization file, that value of interrupt affects all user-defined commands which do not redefine interrupt. Note that built-in FMLI commands (such as **checkworld**) cannot be interrupted. |
| name | The name descriptor defines a string (the name of the command) that will appear in the Command Menu and that users can enter on the command line. |
| oninterrupt | The oninterrupt descriptor specifies the action to be taken when an interrupt signal is received. If it is not defined anywhere in your application, it defaults to `message Operation interrupted!`nop. The oninterrupt descriptor is ignored if interrupt is not coded or if interrupt evaluates to FALSE.<br><br>If defined among the general application descriptors in the initialization file, that value of oninterrupt affects all user-defined commands which do not redefine oninterrupt. |

### Example of Adding an Application-specific Command

You might add a new, application-specific command as follows:

```
name="date"
action=`date | message`nop
help=open $MYFRAMES/Text.datehelp
```

That will give the user a **date** command that puts the current date and time on the message line.

### Example of Disabling an Existing FMLI Command

You can disable an existing FMLI command, **`unix-system`**, for instance, by specifying

```
name=“unix-system”
action=nop
```

When an FMLI command that appears in the `Command Menu` by default is disabled in this way, its name no longer appears in the `Command Menu`.

**NOTE**

When you disable an FMLI command in the commands file, the command becomes unavailable not only to users, but to developers. That is, you cannot use that command in frame definition files or application level files. In particular, do not disable the **`exit`** command.

The contents of the commands file will be reflected in the `Command Menu`. You should avoid giving a command a name that is a partial match of another command, as `pr` is a partial match of both **`prev-frm`** and **`prevpage`**, because this makes it more difficult for users to navigate to (select) that command using character matching.

## The Alias File

The alias file is a file that contains lines of the form

*alias*=*value*

where *alias* is a name to which you have assigned a path name to a file or a device. There are two reasons for having an alias file:

- to simplify references to files or devices with lengthy path names
- to define search paths (similar to `$PATH` in the UNIX system shell)

The name of the alias file must be given as an argument on the **`fmli`** command line with the **`-a`** *alias_file* option when it is invoked.

### Examples of Adding Path Aliases

Whenever you reference a path name that does not begin with a `/` or a `$,` FMLI will check the alias file. For example:

```
MYTEXT=$HOME/myfiles/mytext
```

would allow the developer to refer to the text file **`Text.file`** in the directory **`$HOME/myfiles/mytext`** as **`MYTEXT/Text.file`**.

The alias may also contain the name of the file or device, for example,

```
MYTEXT1=$HOME/myfiles/mytext/Text.file
```

but frame definition file names assigned to an alias must conform to the same naming convention as file names on the invocation line.

More than one possible path may be assigned to a single alias by separating each path with a colon (:). For example:

```
MYFILES=$HOME/myfiles:$HOME/test/myfiles
```

would search **`$HOME/myfiles`** first, and if the file is not found search **`$HOME/test/myfiles`** whenever the alias **`$MYFILES`** is used. This is similar to the way `$PATH` is searched in the UNIX system.

## fmli Command Syntax

The executable file **`fmli`** invokes the Form and Menu Language Interpreter and opens the file(s) you have named as the initial frame definition file(s) to open. It requires at least one argument: an initial frame to open. Subsequent interactions are driven by this initial frame.

The syntax of the **`fmli`** command is as follows:

```
fmli [-i initialization_file] [-c commands_file] [-a alias_file] file ...
```

where *file* is the path name of a frame definition file describing the frame(s) to be opened initially. The argument *file* must follow the file naming convention `Menu.`*xxx* for a menu definition file, `Form.`*xxx* for a form definition file, and `Text.`*xxx* for a text frame definition file, where *xxx* is any string that conforms to UNIX system file naming conventions. The descriptor `lifetime` will be ignored for all frames opened by argument to **`fmli`**. Such frames have a lifetime of `immortal` by default.

Optionally, you may provide the names of *initialization_file*, *commands_file*, and *alias_file*. The *initialization_file* provides specific global instructions that allow for customization of the application, such as redefining screen colors or the default labels and functions assigned to SLKs. The *commands_file* allows the definition of commands specific to your application. The *alias_file* provides access to files via a shell-like (`$PATH`) notation, and allows you to define short, easy-to-use aliases for long path names to files.

**NOTE**

FMLI does not use the end-of-file marker to determine when to exit an application; it uses the FMLI **`exit`** command. For this reason, it is strongly advised that input to FMLI or FMLI applications not be from a pipe ( **|** ), a redirected file ( **<** ), or a here document ( **<<** ).

# 5 Introduction to ETI

# 5 Introduction to ETI

## Overview

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

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

Because the routines are compiled C functions, you should be familiar with the C programming language before using ETI. You should also be familiar with the UNIX system/C language standard I/O package (see the **`stdio(3S)`** manual page) With that knowledge and an appreciation for the philosophy of building on the work of others, you can design screen management programs for many purposes.

## What Is ETI?

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

### The ETI Libraries

ETI consists of

- the low-level (**`curses`**) library
- the `panel` library
- the `menu` library
- the `form` library
- the TAM transition library

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

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

```
ETI is a great package for creating forms and menus.
```

to read

```
ETI is the best package for creating forms and menus.
```

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

Cursor optimization takes care of updating the screen in a manner appropriate for the terminal 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 **`terminfo`** database (described below) to find the correct description for a terminal.

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

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

```
#include <curses.h>

main()
{
   initscr();

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

**Screen 5-1.  A Simple ETI Program**

## The ETI/terminfo Connection

**`terminfo`** is both a set of routines that make use of the capabilities of a wide range of terminals and a database that contains descriptions of the terminals that can be used with ETI. Its use as a database is our concern here. See Chapter 13 for details on its use as a set of routines.

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

Suppose, for instance, that you are using a Teletype 5425 terminal to run the simple ETI program shown in Figure 5-1. To execute properly, the program needs to know how many lines and columns the terminal screen has to print the `BullsEye` in the middle of it. The description of the Teletype 5425 in the terminfo database has this information, as well as other information about the terminal's capabilities and how it performs various operations — for example, how its control characters are interpreted. All ETI needs to know before it goes looking for the information is the name of your terminal.

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

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

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

The first line names the terminal type, and the second line exports it. (See **`profile(4)`**.) The third line of the example tells the UNIX system to initialize the current terminal. That is, it makes sure that the terminal is set up according to its description in the **`terminfo`** database. (The order of these lines is important. `$TERM` must be defined and exported first, so that when **`tput(1)`** is called the proper initialization for the current terminal takes place.) If you had these lines in your **`.profile`** and you ran an ETI program, the program would get the information that it needs about your terminal from the file **`/usr/share/lib/terminfo/5/5425`** in the database, which provides a match for `$TERM`. For more information about the **`terminfo`** database, see Chapter 13 in this guide.

## Other Components of the Screen Management System

You have been given a brief look at the main components of screen management. This section will complete the overview by making you familiar with the other components of this system.

**Figure 5-1. Components of the Screen Management System**

| Component | Brief Description |
|---|---|
| **`terminfo`** database | Files found under **`/usr/share/lib/terminfo/?/*`**; these files contain compiled terminal descriptions. **`?`** is the first letter of the terminal name, and **`*`** is the terminal name. |
| **`tic(1M)`** | **`terminfo(4)`** defines terminal description source files. **`tic`** compiles them into **`terminfo`** database files. |
| **`infocmp(1M)`** | A routine that prints and compares compiled **`terminfo`** description files. |
| **`captoinfo(1M)`** | A routine that converts old **`termcap`** files to **`terminfo`** database files. |
| **`terminfo(4)`** | Defines both the **`terminfo`** database files and the routines used to manipulate and instantiate the strings of data in those files. |
| **`tput(1)`** | A **`terminfo`** routine that causes a string from the **`terminfo`** database to be sent to the terminal, thus setting one or more parameters. |
| **`curses(3curses)`** | A library of C routines that uses information in the **`terminfo`** database. The routines are terminal independent. They optimize cursor movement and allow for the easy programming of screen handling code. |
| Other manual pages to read | **`layers(1), stdio(3S), profile(4), scr_dump(4), term(4), term(5)`** |

The **`terminfo`** database has already been described as one of the main components of the screen management system. The rules for creating a terminal description source file are in the manual page **`terminfo(4)`**. The source file is then compiled using **`tic`**. Unless you have created a shell environment variable called `TERMINFO` that indicates a different path, **`tic`** will place the compiled description file into the proper directory under **`/usr/share/lib/terminfo`** (provided that you have permission to create or overwrite files in that directory). To use **`tic`** simply type:

```
tic filename
```

You may use the **-v** option to get a running commentary. An integer from 1 to 10 may follow the option (no space) to set the level of verbosity. The default
is 1.

The system uses the shell environment variable TERMINFO to find the terminal description files. Initializing a terminal will cause TERMINFO to be set to null and then be converted to **/usr/share/lib/terminfo** unless you have already set it to some other path (**$HOME/bin**, for example). The system will look for the definition of a specific terminal under **$TERMINFO/?/***, where **?** is the first letter of the terminal name, and ***** is the terminal name.

Once a terminal description file has been compiled, it is no longer human readable. The routine **infocmp** translates a compiled description file back to source statements. Invoking the command without arguments will print out the description of the terminal defined by the shell environment variable TERM. A single argument is taken as the name of a terminal you want to see the source description for. With no options declared (or **-I**), you will see descriptions as defined in **terminfo(4)**. There are options for seeing the C variable names (**-L**), the old termcap names (**-C**), and all output in termcap format (**-r**).

If two arguments are given, **infocmp** assumes they identify two descriptions you want to compare. If no options are given (or **-d**), the differences are printed. You may also ask for a list of capabilities that the two have in common (**-c**) or a list of capabilities that neither describes (**-n**). In all of the above cases, the output lists the Boolean fields first, the numeric fields second, and the strings third.

**infocmp** also has options to print, trace, sort, compare files in two different directories, and output a source file derived from the union of two or more compiled description files. For more information consult the **infocmp** manual page.

Early versions of the UNIX system used a different method of describing terminals, called termcap. You can convert a termcap file to a **terminfo** file by using **captoinfo**. If the command is invoked with no arguments, the shell environment variable TERMCAP is used to get the path and the shell environment variable TERM to get the terminal. If TERMCAP is null, the routine tries to convert **/usr/share/lib/termcap**. If a file name is given as an option, that is the file that will be converted. The output is to standard out, and may be piped. Options include a trace mode (**-v**), one field to a line output (**-l**), and changing the output width (**-w**).

One of the definitions given earlier for **terminfo** was that it is a group of routines within **curses** that allow you to manipulate the data in a terminal description file. This small library of routines is documented in this guide and in the **curses(3curses)** manual pages. The command **tput(1)** will allow you to perform many of these manipulations from the command line or in a shell script.

**tput** can always be given the **-T***terminaltype* option, but doesn't need it if the shell environment variable TERM is set. It can be given init, reset, or longname as special arguments. These initialize, reset, and print out the name of the terminal, respectively. Finally, you can use the name of a **terminfo(4)** terminal attribute or capability (called a (*capname*) as an argument. These capabilities can fall into three categories; Boolean, numeric, and strings. If the (*capname* you specify is a string, you may include, as an argument, a list of parameters to insert into coded places in the string (instantiation).

This completes the overview of the screen management system. More detailed information starts in the next chapter. If you elect to skip this and go directly to the manual pages, remember that the examples at the end of the guide might still prove useful.

# 6
# Basic ETI Programming

# 6
# Basic ETI Programming

## Introduction

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

## What Every ETI Program Needs

All ETI programs need to include the header file **`curses.h`** and call the routines `initscr`, `refresh`, or similar routines, and `endwin`. Some of the other header files, however, include **`curses.h`**.

### The Header Files

The header files **`menu.h`**, **`form.h`**, and **`panel.h`** define several global variables and data structures and defines several ETI routines as macros.

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

**NOTE**

`LINES` and `COLS` are external (global) variables that represent the size of a terminal screen. Two similar variables, `$LINES` and `$COLUMNS`, may be set in a user's shell environment; an ETI program uses the environment variables to determine the size of a screen. Whenever we refer to the environment variables in this chapter, we will use the `$`to distinguish them from the C declarations in the **`curses.h`** header file.

For more information about these variables, see "The Routines initscr, refresh, endwin" on page 6-2 and "More about initscr and Lines and Columns" on page 6-5.

The integer variables `COLORS` and `COLOR_PAIRS` are also defined in **`curses.h`**. These will be assigned, respectively, the maximum number of colors and color-pairs the terminal can support. These variables are initialized by the `start_color` routine. (See "Color Manipulation" on page 7-14.)

The header files define the integer constants `OK`, `E_OK`, `ERR` (`E_OK` is in **`eti.h`**), and others listed in the following chapters. ETI routines that return `int` values return these constants under the following conditions:

| | |
|---|---|
| `OK` | returned if a low-level or panel function completes properly |
| `E_OK` | returned if a menu or form function does so |
| `ERR` | returned if a low-level function encounters an error |

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

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

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

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

**CAUTION**

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

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

## The Routines initscr, refresh, endwin

The routines `initscr`, `refresh`, and `endwin` initialize a terminal screen to an "in ETI state," update the contents of the screen, and restore the terminal to an "out of ETI state," respectively. Consider the simple program introduced earlier and reproduced in Screen 6-1.

```
#include <curses.h>

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

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

**Screen 6-1. The Purposes of initscr, refresh, and endwin in a Program**

An ETI program usually starts by calling `initscr`; your program should call `initscr` only once. This routine uses the environment variable `$TERM` to determine what terminal is being used. (See "The ETI/terminfo Connection" on page 5-3 for details.) It then initializes all the declared data structures and other variables from **`curses.h`**. For example, `initscr` would initialize `LINES` and `COLS` for the sample program on whatever terminal it was run. If the Teletype 5425 were used, this routine would initialize `LINES` to 24 and `COLS` to 80. Finally, this routine writes error messages to `stderr` and exits if errors occur.

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

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

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

```
addstr("Bulls");
```

says to write the character string `Bulls`. For example, if the Teletype 5425 were used, these routines would position the cursor and write the character string at (11,36).

**NOTE**

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

Routines like `move` and `addstr` do not actually change a physical terminal screen when they are called. The screen is updated only when `refresh` is called after one or more windows (internal representations of the screen) are updated. This is a very important concept, which we discuss under “More about refresh and Windows” on page 6-5.

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

## Compiling an ETI Program

You compile programs that include ETI routines as C language programs. This means that you use the **`cc(1)`** command to invoke the C compiler. (See the Concurrent *C Reference Manual* for details).

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

The general command line for compiling an ETI program follows:

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

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

### Using the TAM Transition Library

Some users may have applications using the TAM library routines that originally ran on the UNIX PC. Appendix C of this document, explains how to compile and run these applications on any machine of the 3B2 computer family.

## Running an ETI Program

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

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

For an explanation of these lines, turn again to the section “The ETI/terminfo Connection” on page 5-3. Users of an ETI program could also define the environment variables `$LINES`, `$COLUMNS`, and `$TERMINFO` in their **`.profile`** files. However, unlike `$TERM`, these variables do not have to be defined.

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

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

Third, setting break points on ETI routines that are macros, such as `refresh`, does not work. You have to use the routines defined for these macros, instead; for example, you have to use `wrefresh` instead of `refresh`. See “The Header Files” on page 6-1 for more information about macros.

## More about initscr and Lines and Columns

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

## More about refresh and Windows

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

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

**curses.h** supplies a default window named `stdscr` (standard screen), which is the size of the current terminal's screen, for all programs using ETI routines. The header file defines `stdscr` to be of the type `WINDOW*`, a pointer to a C structure which you might think of as a two-dimensional array of characters representing a terminal screen. The program always keeps track of what is on the physical screen, as well as what is in `stdscr`. When `refresh` is called, it compares the two screen images and sends a stream of characters to the terminal that make the physical screen look like `stdscr`. An ETI program considers many different ways to do this, taking into account the various capabilities of the terminal and similarities between what is on the screen and what is on the window (`stdscr`). It optimizes output by printing as few characters as is possible. Figure 6-1 and Figure 6-2 illustrate what happens when you execute the sample ETI program that prints

`BullsEye` at the center of a terminal screen. Notice in the figure that the terminal screen retains whatever garbage is on it until the first `refresh` is called. This `refresh` clears the screen and updates it with the current contents of `stdscr`.

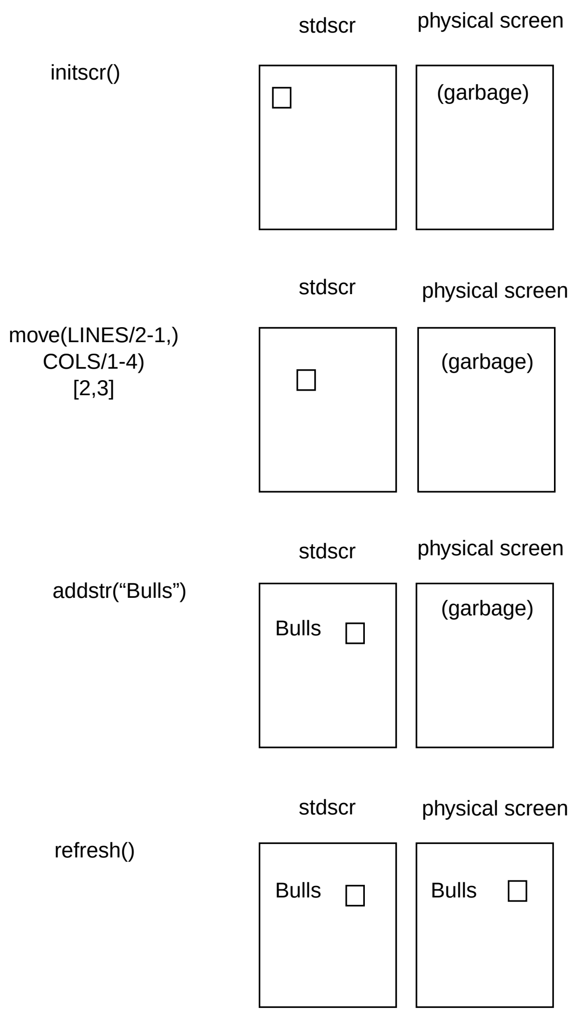


**Figure 6-1. The Relationship between stdscr and a Terminal Screen (Sheet 1 of 2)**

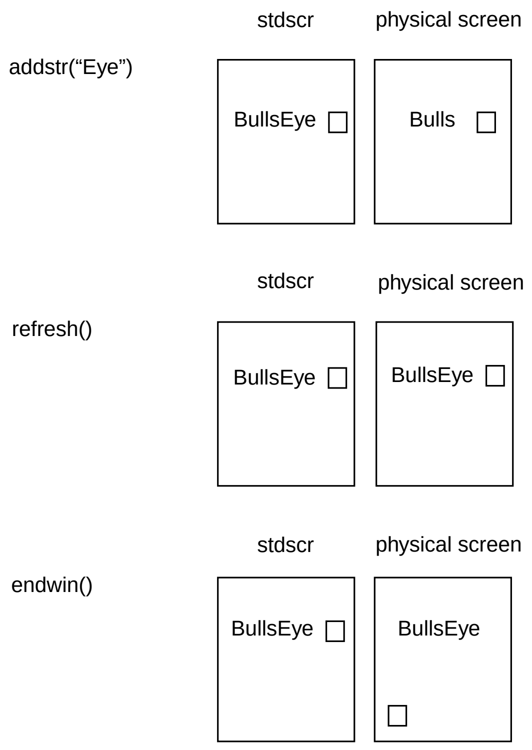


**Figure 6-2. The Relationship between stdscr and a Terminal Screen (Sheet 2 of 2)**

You can create other windows and use them instead of `stdscr`. Windows are useful for maintaining several different screen images. For example, many data entry and retrieval applications use two windows: one to control input and output and one to print error messages that don't mess up the other window. It's possible to subdivide a screen into many windows, refreshing each one of them as desired. And it's possible to create a window within a window; the smaller window is called a subwindow. See Chapter 8 for more information.

## Pads

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

Figure 6-3 represents what a pad, a subwindow, and some other windows could look like in comparison to a physical screen.

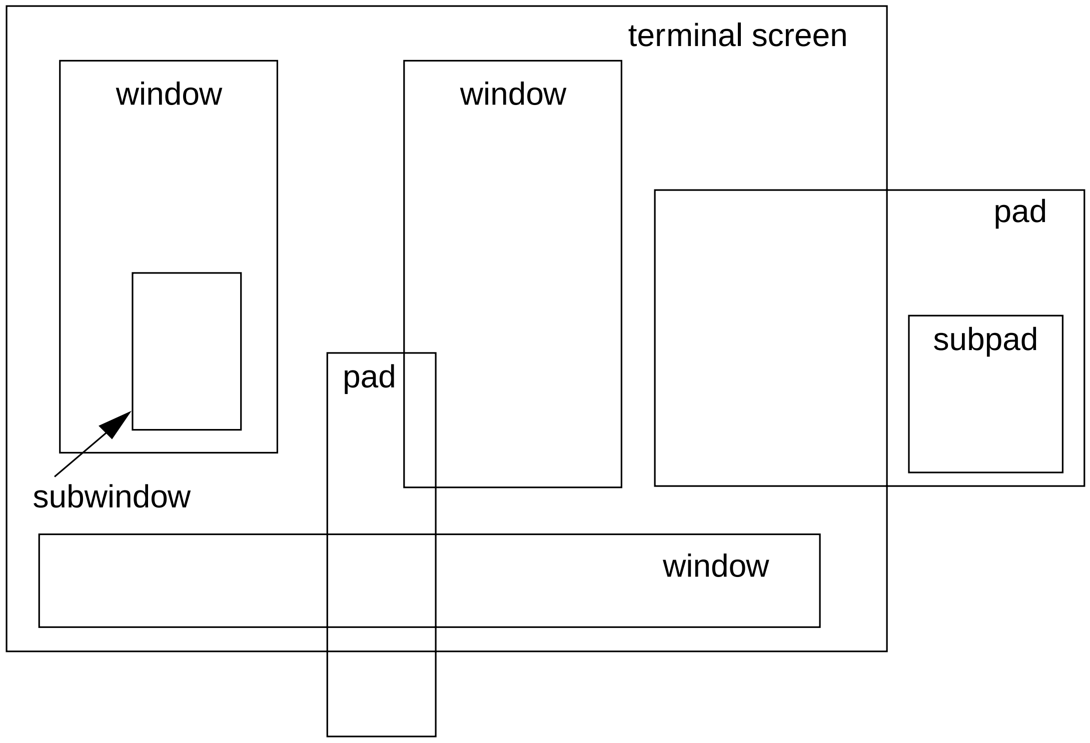


**Figure 6-3. Multiple Windows and Pads Mapped to a Physical Screen**

Chapter 8 describes the routines you use to create and use windows and pads. If you'd like to see an ETI program with windows now, turn to the **`window`** program in Appendix D of this document.

# 7
# Simple Input and Output

# 7
# Simple Input and Output

## Introduction

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

## Output

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

- write a character at a time — `addch`
- write a string — `addstr`
- format a string from a variety of input arguments — `printw`
- move a cursor or move a cursor and print character(s) — `move`, `mvaddch`, `mvaddstr`, `mvprintw`
- clear a screen or a part of it — `clear`, `erase`, `clrtoeol`, `clrtobot`

Following are descriptions and examples of these routines.

**CAUTION**

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

## addch

### SYNOPSIS

```
#include <curses.h>
int addch(ch)
chtype ch;
```

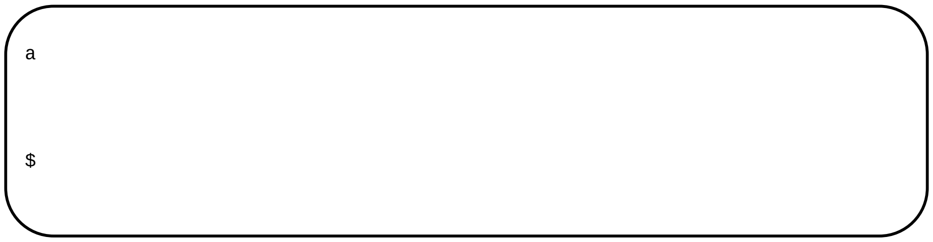
**NOTES**

- `addch` writes a single character to `stdscr` and advances the cursor to the next character position.
- The character is of the type `chtype`, which is defined in **`curses.h`**. `chtype` contains data and attributes (see “Output Attributes” on page 7-11 for information about attributes).
- When working with variables of this type, make sure you declare them as `chtype` and not as the basic type (for example, unsigned `long`) that `chtype` is declared to be in **`curses.h`**. This will ensure future compatibility.
- `addch` does some translations. For example, it converts
  - the `<NL>` character to a clear to end of line and a move to the next line
  - the tab character to an appropriate number of blanks
  - other control characters to their `^`*X* notation
- `addch` normally returns `OK`. The only time `addch` returns `ERR` is after adding a character to the lower right-hand corner of a window that does not scroll.
- `addch` is a macro.

**EXAMPLE**

```
#include <curses.h>
main()
{
   initscr();
   addch('a');
   refresh();
   endwin();
}
```

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



```
a



$
```

See also the **`show`** program in Appendix D of this document.

## addstr

### SYNOPSIS

```
#include <curses.h>
int addstr(str)
char *str;
```

### NOTES

- `addstr` writes a string of characters to `stdscr`.
- `addstr` calls `addch` to write each character.
- `addstr` follows the same translation rules as `addch`.
- `addstr` returns `OK` on success and `ERR` on error.
- `addstr` is a macro.

### EXAMPLE

Recall the sample program that prints the character string `BullsEye`. See Screen 6-1, Figure 6-1, and Figure 6-2.

## printw

### SYNOPSIS

```
#include <curses.h>
int printw(fmt [,arg...])
char *fmt;
```

### NOTES

- `printw` handles formatted printing on `stdscr`.
- Like `printf`, `printw` takes a format string and a variable number of arguments.
- Like `addstr`, `printw` calls `addch` to write the string.
- `printw` returns `OK` on success and `ERR` on error.

### EXAMPLE

```
#include <curses.h>
main()
{
    char* title = “Not specified”;
    int no = 0;
```

```
            /* Missing code. */

        initscr();

            /* Missing code. */

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

        refresh();
        endwin();
    }
```

The output from this program will appear as follows:

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



$
```

## move

### SYNOPSIS

```
#include <curses.h>
int move(y, x)
int y, x;
```

### NOTES

- `move` positions the cursor for `stdscr` at the given row *y* and the given column *x*.
- Notice that `move` takes the *y* coordinate before the *x* coordinate. The upper left-hand coordinates for `stdscr` are (0,0), the lower right-hand (`LINES - 1, COLS - 1`). See the section “The Routines initscr, refresh, endwin” on page 6-2 for more information.
- `move` may be combined with the write functions to form
  - `mvaddch(` *y*, *x*, *ch*`)`, which moves to a given position and prints a character
  - `mvaddstr(` *y*, *x*, *str*`)`, which moves to a given position and prints a string of characters

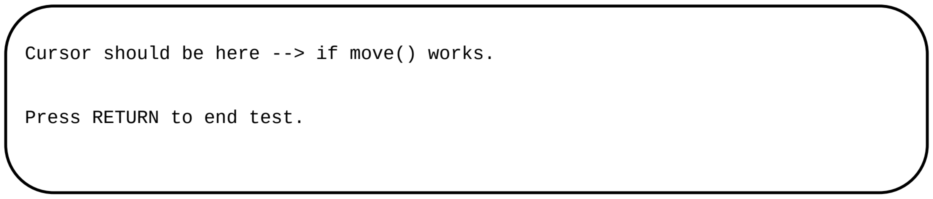
  - `mvprintw(` *y*`,` *x*`,` *fmt* [*,arg...*]`)`, which moves to a given position and prints a formatted string.

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

**EXAMPLE**

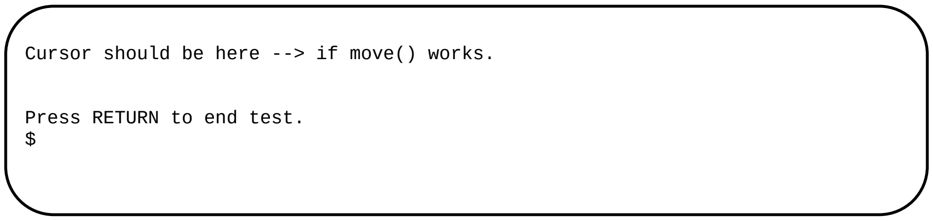
```
#include <curses.h>
main()
{
   initscr();
   addstr(“Cursor should be here --> if move() works.”);
   printw(“\n\n\nPress RETURN to end test.”);
   move(0,25);
   refresh();
   getch();     /* Gets RETURN; discussed below. */
   endwin();
}
```

Here's the output generated by running this program:



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


Press RETURN to end test.
```

After you press RETURN, the screen looks like this:



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


Press RETURN to end test.
$
```

See the **`scatter`** program in Appendix D of this document for another example using `move`.

## clear and erase

**SYNOPSIS**

```
#include <curses.h>
int clear()
int erase()
```

**NOTES**

- Both routines change `stdscr` to all blanks.
- `clear` assumes that the screen may have garbage that it doesn't know about; this routine first calls `erase` and then `clearok` which clears the physical screen completely on the next call to `refresh` for `stdscr`. See the low-level {VS} or **`curses(3curses)`** manual pages for more information about `clearok`.
- `initscr` automatically calls `clear`.
- In ETI UNIX System V Release 3.1 and later releases, `clear` and `erase` always return `OK`.
- Both routines are macros.

## clrtoeol and clrtobot

**SYNOPSIS**

```
#include <curses.h>
int clrtoeol()
int clrtobot()
```

**NOTES**

- `clrtoeol` changes the remainder of a line to all blanks.
- `clrtobot` changes the remainder of a screen to all blanks.
- Both begin at the current cursor position inclusive.
- Neither returns any useful value.

**EXAMPLE**

```
#include <curses.h>
main()
{
   initscr();
   addstr(“Press RETURN to delete from here to the end
       of the line and on.”);
   addstr(“\nDelete this too.\nAnd this.”);
```

```
        move(0,30);
        refresh();
        getch();
        clrtobot();
        refresh();
        endwin();
    }
```

Here's the output generated by running this program:

```
Press RETURN to delete from here to the end of the line and on.
Delete this too.
And this.
```

Notice the two calls to `refresh`: one to send the full screen of text to a terminal, the other to clear from the position indicated to the bottom of a screen. Here's what the screen looks like when you press `RETURN`:

```
Press RETURN to delete from here

$
```

See the **`show`** and **`two`** programs in Appendix D of this document for other uses of `clr-toeol`.

## Input

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

- read a character at a time — `getch`
- read a `<NL>`-terminated string — `getstr`
- parse input, converting and assigning selected data to an argument list — `scanw`

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

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

## getch

### SYNOPSIS

```
#include <curses.h>
int getch()
```

### NOTES

- `getch` reads a single character from the current terminal.
- `getch` returns the value of the character or `ERR` on end of file, receipt of signals, or non-blocking read with no input.
- `getch` is a macro.
- See the discussions of `echo`, `noecho`, `cbreak`, `nocbreak`, `raw`, `noraw`, `halfdelay`, `nodelay`, and `keypad` below and in **`curses(3curses)`**.

### EXAMPLE

```
#include <curses.h>
main()
{
   int ch;

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

The output from this program follows. The first `refresh` sends the `addstr` character string from `stdscr` to the terminal:

```
Press any character:
```

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

```
Press any character:  w

The character entered was a 'w'.

$
```

For another example of `getch`, see the **`show`** program in Appendix D of this document.

## getstr

### SYNOPSIS

```
#include <curses.h>
int getstr(str)
char *str;
```

### NOTES

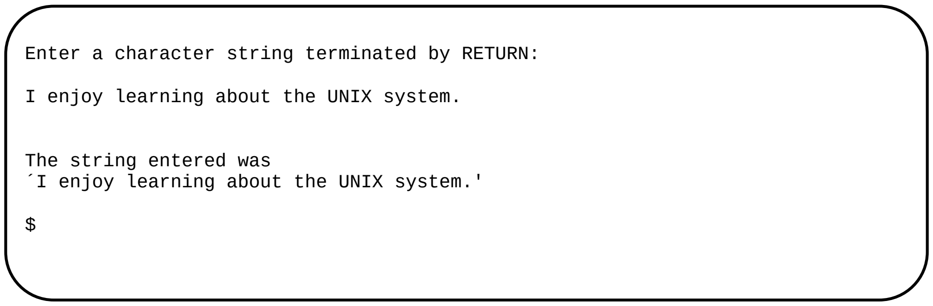
- `getstr` reads characters and stores them in a buffer until a `RETURN`, `<NL>`, or `<ENTER>` is received from `stdscr`. `getstr` does not check for buffer overflow.
- The characters read and stored are in a character string.
- `getstr` is a macro; it calls `getch` to read each character.
- `getstr` returns ERR if `getch` returns ERR to it. Otherwise it returns `OK`.
- See the discussions of `echo, noecho, cbreak, nocbreak, raw, noraw, halfdelay, nodelay,` and `keypad` below and in ETI **`curses(3curses)`**.

### EXAMPLE

```
#include <curses.h>
main()
{
```

```
char str[256];

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

Assume you entered the string `'I enjoy learning about the UNIX system'`. The final screen (after entering `RETURN`) would appear as follows:



```
Enter a character string terminated by RETURN:

I enjoy learning about the UNIX system.


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

$
```

## scanw

### SYNOPSIS

```
#include <curses.h>
int scanw(fmt [, arg...])
char *fmt;
```

### NOTES

- `scanw` calls `getstr` and parses an input line.
- Like **`scanf(3S)`**, `scanw` uses a format string to convert and assign to a variable number of arguments.
- `scanw` returns the same values as `scanf`.
- See **`scanf(3S)`** for more information.

### EXAMPLE

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

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

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

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

```
The string was “twin” and the number was 2.000000.

$
```

## Output Attributes

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

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

- `A_BLINK` — blinking
- `A_BOLD` — extra bright or bold
- `A_DIM` — half bright
- `A_REVERSE` — reverse video

- A_STANDOUT — a terminal's best highlighting mode
- A_UNDERLINE — underlining
- A_ALTCHARSET — alternate character set (see “Routines for Drawing Lines and Other Graphics” on page 12-1)

(See “Color Manipulation” on page 7-14 for information on using colors.)

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

**NOTE**

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

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

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

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

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

In addition to the attributes listed above, there are two bit masks called A_CHARTEXT and A_ATTRIBUTES. You can use these bit masks with the ETI function inch and the C logical AND ( & ) operator to extract the character or attributes of a position on a terminal screen. See the discussion of inch on the **curses(3curses)** manual pages.

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

## attron, attrset, and attroff

#### SYNOPSIS

```
#include <curses.h>
int attron( attrs )
chtype attrs;
int attrset( attrs )
chtype attrs;
int attroff( attrs )
chtype attrs;
```

#### NOTES

- `attron` turns on the requested attribute `attrs` in addition to any that are currently on. `attrs` is of the type `chtype` and is defined in **`curses.h`**.
- `attrset` turns on the requested attributes `attrs` instead of any that are currently turned on.
- `attroff` turns off the requested attributes `attrs` if they are on.
- The attributes may be combined using the bitwise OR ( `|` ).
- All return 1 (not `OK`).

#### EXAMPLE

See the **`highlight`** program in Appendix D of this document.

## standout and standend

#### SYNOPSIS

```
#include <curses.h>
int standout()
int standend()
```

#### NOTES

- `standout` turns on the preferred highlighting attribute, `A_STANDOUT`, for the current terminal. This routine is equivalent to `attron(A_STANDOUT)`.
- `standend` turns off all attributes. This routine is equivalent to `attr-set(0)`, where `attrset` takes the argument `0`.
- Both always return 1 (not `OK`).

#### EXAMPLE

Again, see the **`highlight`** program in Appendix D of this document.

## Color Manipulation

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

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

### How the Color Feature Works

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

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

**NOTE**

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

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

**Table 7-1. The Default Colors Table**

| | Intensity of Component | | |
|---|---|---|---|
| | (R)ed | (G)reen | (B)lue |
| /* black: 0 */ | 0 | 0 | 0 |
| /* red: 1 */ | 1000 | 0 | 0 |
| /* green: 2 */ | 0 | 1000 | 0 |

**Table 7-1. The Default Colors Table**

| | Intensity of Component | | |
|---|---|---|---|
| | (R)ed | (G)reen | (B)lue |
| /* yellow: 3 */ | 1000 | 1000 | 0 |
| /* blue: 4 */ | 0 | 0 | 1000 |
| /* magenta: 5 */ | 1000 | 0 | 1000 |
| /* cyan: 6 */ | 0 | 1000 | 1000 |
| /* white: 7 */ | 1000 | 1000 | 1000 |

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

You can change these color definitions with the routine `init_color`, if your terminal is capable of redefining colors. (See “Changing the Definitions of Colors” on page 7-17 for more information.)

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

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

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

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

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

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

**Table 7-2. Example of a Pairs Table**

| Color-Pair Number | Foreground | Background |
|---|---|---|
| 0 (reserved) | 0 | 0 |
| 1 | 1 | 6 |
| 2 | 3 | 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.

Upon termination of your **`curses`** program, all colors and/or color-pairs will be restored to the values they had when the terminal was just turned on.

## Using the COLOR_PAIR(*n*) Attribute

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

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

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

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

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

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

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

would turn on blinking and whatever you have initialized color-pair 1 to be. (`attron` and `attrset` are described earlier in this chapter and also on the **`curses(3curses)`** manual pages in this guide.

## Changing the Definitions of Colors

If your terminal is capable of redefining colors, you can change the predefined colors with the routine `init_color(`*color, r, g, b*). The first argument, *color*, is the numeric value of the color you want to change, and the last three, *r*, *g*, and *b*, are the intensities of the red, green, and blue components, respectively, that the new color will contain. Once you change the definition of a color, all occurrences of that color on your screen change immediately.

So, for example, you could change the definition of color 4 (`COLOR_BLUE` by default), to be light blue, in the following way.

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

If your terminal is not able to change the definition of a color, use of `init_color` returns ERR.

## Portability Guidelines

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

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

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

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

- Do not use color 0 as a background color.

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

- Combine color and other video attributes.

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

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

## Other Macros and Routines

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

- `A_COLOR` is a bit mask to extract color-pair information. It can be used to clear the color-pair field, and to determine if any color-pair is being used.
- `PAIR_NUMBER(`*attrs*) is the reverse of `COLOR_PAIR`(n). It returns the color-pair number associated with the named attribute, *attrs*.

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

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

These routines are explained in more detail on the **`curses(3curses)`** manual pages in this guide.

The routines `start_color`, `init_color`, and `init_pair` are described on the following pages, with examples of their use. You can also refer to the program **`colors`** in Appendix D for an example of using the attribute of color in windows.

## start_color

### SYNOPSIS

```
#include <curses.h>
int start_color()
```

**NOTES**

- This routine must be called if you want to use colors, and before any other color manipulation routine is called. It is good practice to call it right after `initscr`.
- It initializes eight default colors (black, red, green, yellow, blue, magenta, cyan, and white), and the global variables `COLORS` and `COLOR_PAIRS`. If the value corresponding to `COLOR_PAIRS` in the terminfo database is greater than 64, `COLOR_PAIRS` will be set to 64.
- It restores the terminal's colors to the values they had when the terminal was just turned on.
- It returns `ERR` if the terminal does not support colors, `OK` otherwise.

**EXAMPLE**

See the example under `init_pair`.

## init_pair

**SYNOPSIS**

```
#include <curses.h>
int init_pair (pair, f, b)
short pair, f, b;
```

**NOTES**

- `init_pair` changes the definition of a color-pair.
- Color-pairs must be initialized with `init_pair` before they can be used as the argument to the attribute macro `COLOR_PAIR(`*n*`)`.
- The value of the first argument, *pair*, is the number of a color-pair, and must be between `1` and `COLOR_PAIRS-1`.
- The value of the *f* (foreground) and *b* (background) arguments must be between `0` and `COLORS-1`.
- If the color-pair was previously initialized, the screen will be refreshed and all occurrences of that color-pair will change to the new definition.
- It returns `OK` if it was able to change the definition of the color-pair, `ERR` otherwise.

**EXAMPLE**

```
#include <curses.h>
main()
{
    initscr ();
    if (start_color () == OK)
```

```
        {
            init_pair (1, COLOR_RED, COLOR_GREEN);
            attron (COLOR_PAIR (1));
            addstr (“Red on Green”);
            getch();
        }
        endwin();
    }
```

Also see the program **`colors`** in Appendix D of this document.

## init_color

### SYNOPSIS

```
#include <curses.h>
int init_color(color, r, g, b)
short color, r, g, b;
```

### NOTES

- `init_color` changes the definition of a color.
- The first argument, *color*, is the number of the color to be changed. The value of *color* must be between `0` and `COLORS-1`.
- The last three arguments, *r*, *g*, and *b*, are the amounts of red, green, and blue (RGB) components in the new color. The values of these three arguments must be between `0` and `1000`.
- When `init_color` is used to change the definition of an entry in the colors table, all places where the old color was used on the screen immediately change to the new color.
- It returns `OK` if it was able to change the definition of the color, `ERR` otherwise.

### EXAMPLE

```
#include <curses.h>
main()
{
    initscr();
    if (start_color() == OK)
    {
        init_pair (1, COLOR_RED, COLOR_GREEN);
        attron (COLOR_PAIR (1));
        if (init_color (COLOR_RED, 0, 0, 1000) == OK)
            addstr (“BLUE ON GREEN”);
        else
            addstr (“RED ON GREEN”);
        getch ();
    }
```

```
        endwin();
    }
```

## Bells, Whistles, and Flashing Lights: beep and flash

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

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

### SYNOPSIS

```
    #include <curses.h>
    int flash()
    int beep()
```

### NOTES

- `flash` tries to flash the terminals screen, if possible, and, if not, tries to ring the terminal bell.
- `beep` tries to ring the terminal bell, if possible, and, if not, tries to flash the terminal screen.
- `beep` will not work if you redefine `TRUE` to something other than 1.
- Neither returns any useful value.

## Input Options

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

- echoes (prints back) characters to a terminal as they are typed
- interprets an erase character (typically #) and a line kill character (typically `@`)
- interprets a `CTRL-d` (control d) as end of file (`EOF`)
- interprets interrupt and quit characters
- strips the character's parity bit
- translates RETURN to <NL>

Because an ETI program maintains total control over the screen, low-level ETI turns off echoing on the UNIX system and does echoing itself. At times, you may not want the UNIX system to process other characters in the standard way in an interactive screen management program. Some ETI routines, `noecho` and `cbreak`, for example, have been designed so that you can change the standard character processing. Using these routines in an application controls how input is interpreted. Table 7-3 shows some of the major routines for controlling input.

Every low-level {VS} program accepting input should set some input options. This is because when the program starts running, the terminal on which it runs may be in `cbreak`, `raw`, `nocbreak`, or `noraw` mode. Although the low-level {VS} program starts up in `echo` mode, none of the other modes are guaranteed.

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

**Table 7-3. Input Option Settings for ETI Programs**

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

**CAUTION**

Do not use the combination `nocbreak` and `echo`. If you use it in a program and also use `getch`, the program will go in and out of `cbreak` mode to get each character. Depending on the state of the tty driver when each character is typed, the program may produce undesirable output.

In addition to the routines noted in Table 7-3, you can use the ETI routines `noraw`, `halfdelay`, and `nodelay` to control input. See the **`curses(3curses)`** manual pages for discussions of these routines.

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

## echo and noecho

### SYNOPSIS

```
#include <curses.h>
int echo()
int noecho()
```

### NOTES

- `echo` turns on echoing of characters by ETI as they are read in. This is the initial setting.
- `noecho` turns off the echoing.
- Neither returns any useful value.
- ETI programs may not run properly if you turn on echoing with `nocbreak`. See Table 7-3 and accompanying caution. After you turn echoing off, you can still echo characters with `addch`.

### EXAMPLE

See the **`editor`** and **`show`** programs in Appendix D of this document.

## cbreak and nocbreak

### SYNOPSIS

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

## NOTES

- `cbreak` turns on “break for each character” processing. A program gets each character as soon as it is typed, but the erase, line kill, and CTRL-D characters are not interpreted.
- `nocbreak` returns to normal “line at a time” processing. This is typically the initial setting.
- Neither returns any useful value.
- ETI programs may not run properly if `cbreak` is turned on and off within the same program or if the combination `nocbreak` and `echo` is used.
- See Table 7-3 and accompanying caution.

## EXAMPLE

See the **`editor`** and **`show`** programs in Appendix D of this document.

# 8
# Windows

# 8
# Windows

## Introduction

“More about refresh and Windows” on page 6-5 explained what windows and pads are and why you might want to use them. This section describes the ETI routines you use to manipulate and create windows and pads.

## Output and Input

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

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

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

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

## The Routines wnoutrefresh and doupdate

If you recall from the earlier discussion about `refresh`, we said that it sends the output from `stdscr` to the terminal screen. We also said that it was a macro that expands to `wrefresh(stdscr)` (see “What Every ETI Program Needs” on page 6-1 and “More about refresh and Windows” on page 6-5).

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

Using `wnoutrefresh`—or `pnoutrefresh` (this discussion will be limited to the former routine for simplicity)—and `doupdate`, you can update terminal screens more efficiently than using `wrefresh` by itself. `wrefresh` works by first calling `wnoutrefresh`, which copies the named window to a data structure referred to as the virtual screen. The virtual screen contains what a program intends to display at a terminal. After calling `wnoutrefresh`, `wrefresh` then calls `doupdate`, which compares the virtual screen to the physical screen and does the actual update. If you want to output several windows at once, calling `wrefresh` will result in alternating calls to `wnoutrefresh` and `doupdate`, causing several bursts of output to a screen. However, by calling `wnoutrefresh` for each window and then `doupdate` only once, you can minimize the total number of characters transmitted and the processor time used. Screen 8-1 shows a sample program that uses only one `doupdate`.

```
#include <curses.h>

main()
{
   WINDOW *w1, *w2;

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

**Screen 8-1. Using wnoutrefresh and doupdate**

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

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

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

Figure 8-1, Figure 8-2, and Figure 8-3 illustrate the effect of `wnoutrefresh` and `doupdate` on these two windows, the virtual screen, and the physical screen.

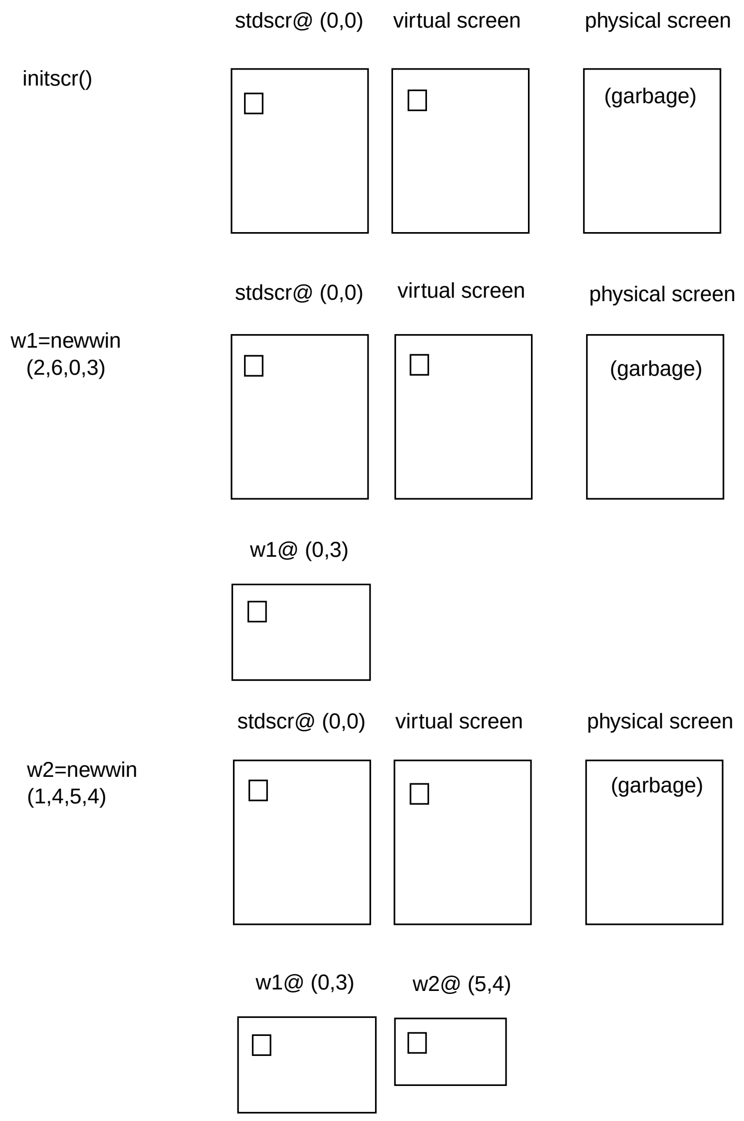


**Figure 8-1. Relationship between a Window and Terminal Screen (Sheet 1 of 3)**

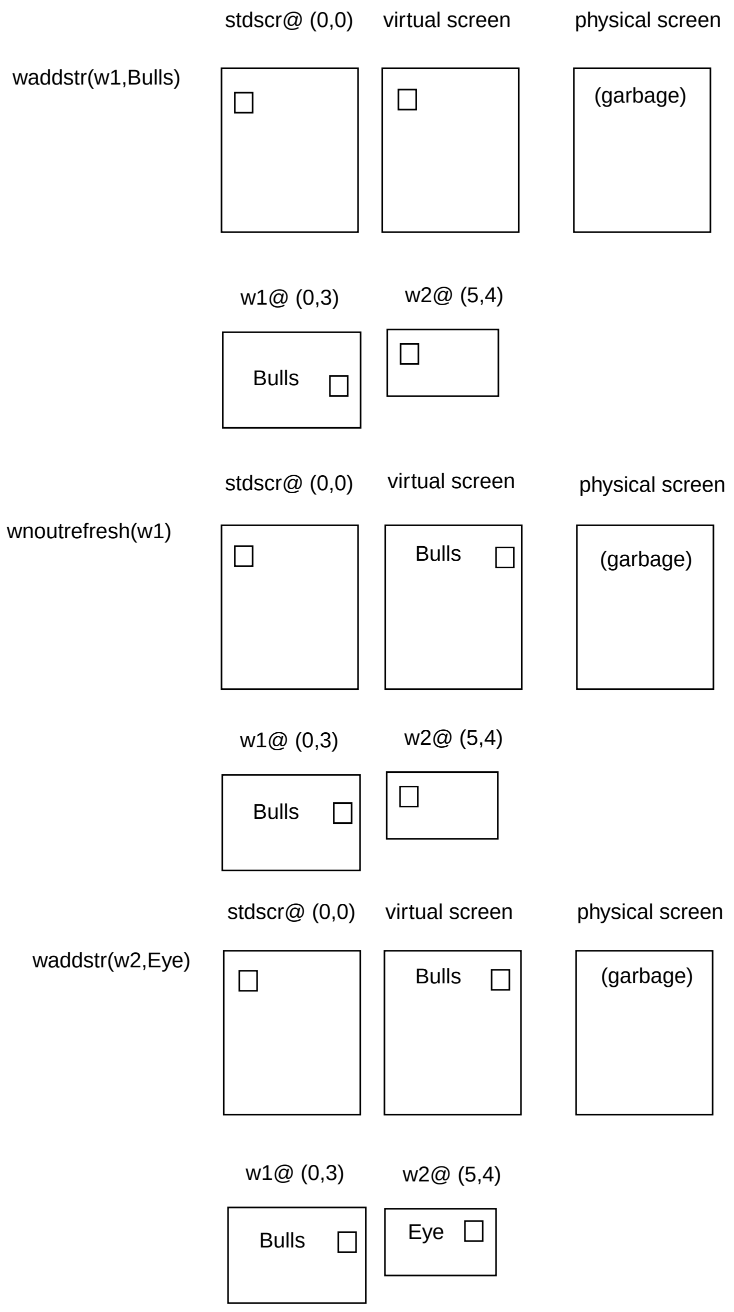


**Figure 8-2. Relationship between a Window and Terminal Screen (sheet 2 of 3)**

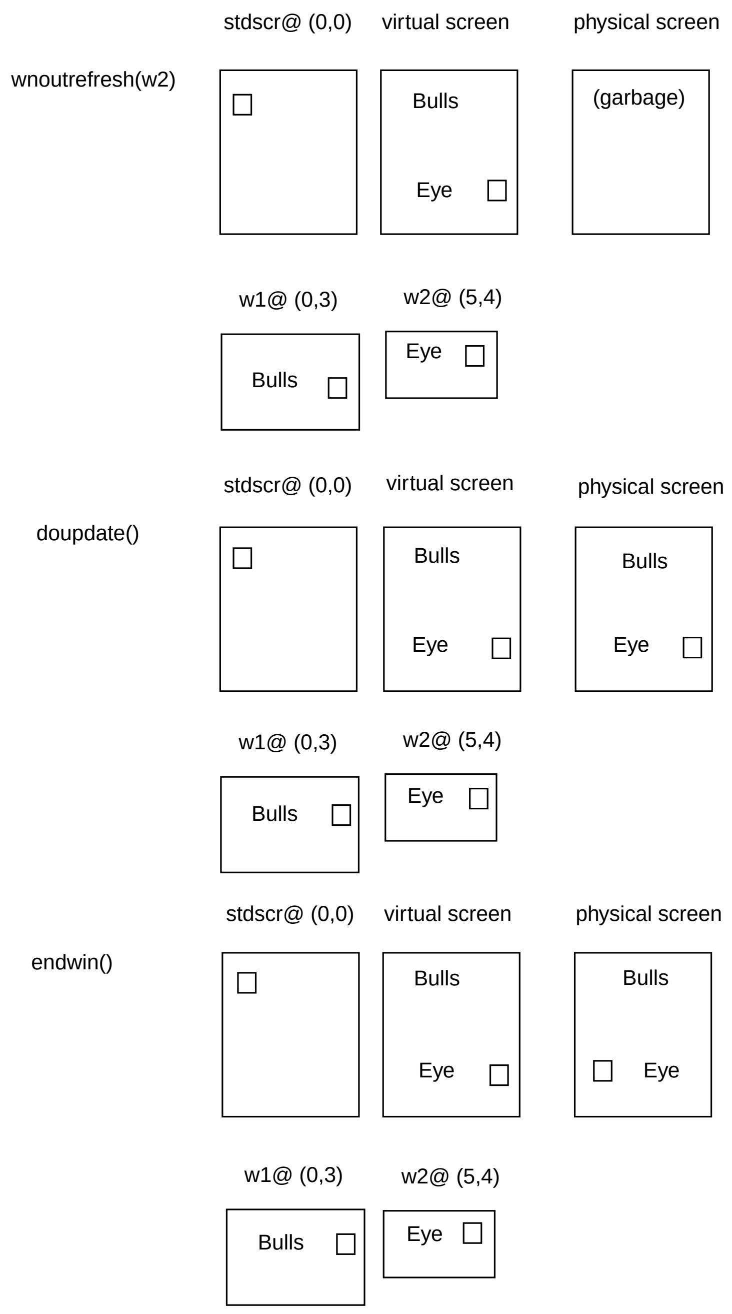


**Figure 8-3. Relationship between a Window and Terminal Screen (sheet 3 of 3)**

## New Windows

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

### newwin

#### SYNOPSIS

```
#include <curses.h>
WINDOW *newwin(nlines, ncols, begin_y, begin_x)
int nlines, ncols, begin_y, begin_x;
```

#### NOTES

- `newwin` returns a pointer to a new window with a new data area.
- The variables *nlines* and *ncols* give the size of the new window.
- *begin_y* and *begin_x* give the screen coordinates from (0,0) of the upper left corner of the window as it is refreshed to the current screen.

#### EXAMPLE

Recall the sample program using two windows; see Screen 8-1. Also see the **`window`** program in Appendix D of this document.

### subwin

#### SYNOPSIS

```
#include <curses.h>
WINDOW *subwin(orig, nlines, ncols, begin_y, begin_x)
WINDOW *orig;
int nlines, ncols, begin_y, begin_x;
```

#### NOTES

- `subwin` returns a new window that points to a section of another window, *orig*.
- *nlines* and *ncols* give the size of the new window.
- *begin_y* and *begin_x* give the screen coordinates of the upper left corner of the window as it is refreshed to the current screen.
- Subwindows and original windows can accidentally overwrite one another.

**CAUTION**

Subwindows of subwindows do not work (as of the copyright date of this guide).

**EXAMPLE**

```
#include <curses.h>

main()
{
 WINDOW *sub;

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

This program prints a border of `w`'s around `stdscr` (the sides of your terminal screen) and a border of `s`'s around the subwindow `sub` when it is run. For another example, see the **`window`** program in Appendix D of this document.

## ETI Low-level Interface (curses) to High-level Functions

In the following chapters, we will consider the ETI high-level functions, which create and manipulate panels, menus, and forms. All application programs that use these high-level functions require a set of low-level ETI (**`curses`**) calls that properly initialize and terminate the programs. For convenience, you may want to isolate these calls in appropriate routines. Screen 8-2 shows one way you might do this. It lists routines to start low-level ETI, terminate it, and handle fatal errors.

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

**Screen 8-2. Sample Routines for Low-level ETI (curses) Interface**

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

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

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

# 9
# Panels

# 9
# Panels

## Introduction

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

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

## Compiling and Linking Panel Programs

To use the panel routines, you specify

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

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

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

## Creating Panels

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

#### SYNOPSIS

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

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

To create a panel, you create a window, save the pointer to it, and use the pointer as an argument to `new_panel`.

```
WINDOW *win;
PANEL *pptr;

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

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

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

## Elementary Panel Window Operations

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

### Fetching Pointers to Panel Windows

Each panel has a low-level ETI window associated with it. To retrieve a pointer to this window, you use the function `panel_window`.

#### SYNOPSIS

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

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

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

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

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

## Changing Panel Windows

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

### SYNOPSIS

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

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

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

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

panel = new_panel(win1);

   /* intervening processing with win1 as panel window */

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

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

## Moving Panel Windows on the Screen

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

**SYNOPSIS**

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

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

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

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

```
PANEL *panel;

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

## Moving Panels to the Top or Bottom of the Deck

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

**SYNOPSIS**

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

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

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

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

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

```
PANEL *panel1, *panel2;

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

## Updating Panels on the Screen

Function update_panels makes all low-level **curses** calls (such as touchwin and wnoutrefresh) to update all panels so as to maintain proper depth relationships and to permit display only of the appropriate portions of panels.

### SYNOPSIS

```
void update_panels();
```

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

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

**CAUTION**

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

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

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

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, you write:

```
    WINDOW *win;

    update_panels();
    doupdate();
```

Note finally that there is no way to display the updates to an obscured panel without displaying the changes to all panels.

## Making Panels Invisible

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

### Hiding Panels

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

**SYNOPSIS**

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

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

**NOTE**

> Only the operations `top_panel`, `bottom_panel`, and `hide_panel` itself 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, you 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, you write

```
    PANEL *panel2;

    hide_panel(panel2);
```

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

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

### Checking If Panels Are Hidden

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

#### SYNOPSIS

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

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

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

```
PANEL *panel;

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

## Reinstating Panels

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

#### SYNOPSIS

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

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

To return, say, `panel2` to the deck, you write

```
PANEL *panel2;

show_panel(panel2);
```

## Fetching Panels above or below Given Panels

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

**SYNOPSIS**

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

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

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

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

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

- they have been hidden using `hide_panel`
- all panels have been deleted
- or no panels have been created.

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

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

## Setting and Fetching the Panel User Pointer

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

**SYNOPSIS**

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

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

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 Screen 9-1, create a hidden panel for pop-up messages.

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

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

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

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

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

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

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

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

    display_deck (show_mess);
```

**Screen 9-1.  Example Using Panel User Pointer**

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

## Deleting Panels

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

**SYNOPSIS**

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

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

**NOTE**

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

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

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

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

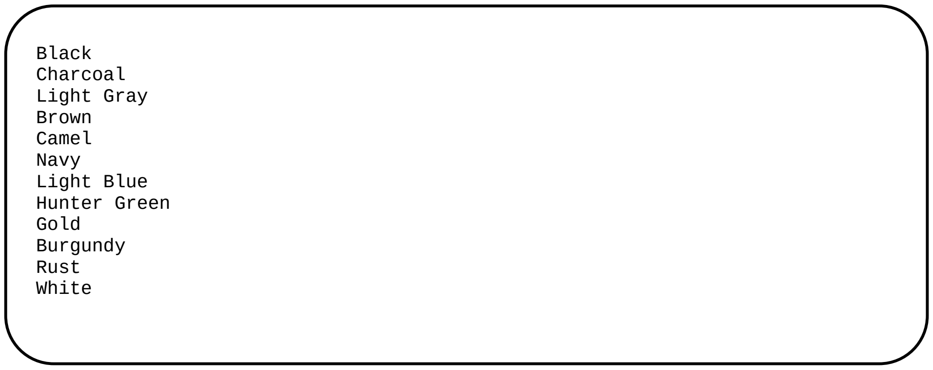
```
    PANEL *panel;
    WINDOW *win = panel_window(panel);

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

# 10 Menus

# 10
# Menus

## Introduction

A menu is a screen display that presents a set of items from which the user selects one or more, depending on the type of menu. Once the user makes a selection, your application program responds accordingly. This response may be to generate a message, display another menu, or take some other action. Screen 10-1 displays a sample menu.



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

**Screen 10-1. A Sample Menu**

## Compiling and Linking Menu Programs

To use the menu routines, you specify

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

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

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

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

# Overview: Writing Menu Programs in ETI

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

## Some Important Menu Terminology

The following terms will be helpful:

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

## What a Menu Application Program Does

In general, a menu application program will

- initialize low-level ETI (**`curses`**)
- create the items for the menu
- create the menu
- post the menu
- refresh the screen

- process end user menu requests
- unpost the menu
- free the menu
- free items
- terminate low-level ETI (**`curses`**)

## A Sample Menu Program

Screen 10-2 shows the ETI code necessary for generating the menu of colors in Screen 10-1.

```
#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;

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

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

    /* create items */

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

    /* create and display menu */

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

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

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

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

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

**Screen 10-2. Sample Menu Program to Create a Menu in ETI**

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

Every menu program should have the line

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

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

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

Next, the menu is created and connected to the item pointer array 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 five seconds.

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

Finally, functions `endwin` and `exit` terminate low-level ETI and the program itself.

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

**NOTE**

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

## Creating and Freeing Menu Items

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

**SYNOPSIS**

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

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

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

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

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

In general, you use an array to store the item pointers returned by `new_item`. Screen 10-3 shows how you might create an item array of the planets of our solar system.

```
ITEM * planets[10];

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

**Screen 10-3. Creating an Array of Items**

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

### SYNOPSIS

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

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

The argument to `free_item` is a pointer previously obtained from `new_item`.

**NOTE**

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

Once an item is freed, you must not use it again. If a freed item's pointer is passed to an ETI routine, undefined results will occur.

If successful, `free_item` returns `E_OK.` If it encounters an error, it returns one of the following:

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

## Two Kinds of Menus: Single- or 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 “Setting and Fetching Menu Options” on page 10-47.

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

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

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

### SYNOPSIS

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

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

Function `set_item_value` fails if given an item that is not selectable (the `O_SELECTABLE` option was previously turned off) or the item is connected to a single-valued menu (connecting items to menus is described in “Creating and Freeing Menus” on page 10-13). If successful, `set_item_value` returns `E_OK`. Otherwise, one of the following is returned.

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

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

You might want to place the code in Screen 10-4 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.

```
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;
    }
}
```

**Screen 10-4. Using item_value in Menu Processing**

## Manipulating Item Attributes

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

### Fetching Item Names and Descriptions

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

**SYNOPSIS**

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

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

Both functions return NULL if given a NULL item pointer.

## Setting Item Options

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

### SYNOPSIS

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

OPTIONS item_opts (item)
ITEM * item;

options:
    O_SELECTABLE
```

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

If successful, `set_item_opts` returns `E_OK`. Otherwise, it returns the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |

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

If you turn off option `O_SELECTABLE`, the item cannot be selected. You might want to make an item unselectable to emphasize certain things your application program is doing. Unselectable items are displayed using the grey display attribute, described in “Fetching and Changing a Menu's Display Attributes” on page 10-25.

Because options are Boolean values (they are either on or off), you 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`, you can write:

```
ITEM * i0, * i1;

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

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

**SYNOPSIS**

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

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

These functions return the same error conditions as `set_item_opts`.

As an example, the following code turns option `O_SELECTABLE` off for item `i0` and on for item `i1`.

```
ITEM * i0, * i1;

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

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

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

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

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

or

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

## Checking an Item's Visibility

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

**SYNOPSIS**

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

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

To check if the first menu item is currently visible on the display, you can 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 “Counting the Number of Menu Items” on page 10-16.

### Changing the Current Default Values for Item Attributes

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

**NOTE**

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

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

## Setting the Item User Pointer

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

**SYNOPSIS**

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

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

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

Any defined structure can be connected to an item using the item's user pointer. The pointer must be cast to (char *) and then later recast back to (defined-struct *). Screen 10-5 shows how to use an item's user pointer with a struct `ITEM_ID`, which stores biological information.

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

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

ITEM * items[7];

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

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

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

**Screen 10-5. Using an Item User Pointer**

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

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

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

Finally, you might call `get_type` to write the type, thus:

```
WINDOW * win;

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

If successful, `set_item_userptr` returns `E_OK`. Otherwise, it returns the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |

If function `set_item_userptr` is passed a NULL item pointer, the argument `userptr` becomes the new default user pointer for all subsequently created items. As an 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 itself. To create and initialize a menu, you use function `new_menu`.

**SYNOPSIS**

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

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

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

**NOTE**

> Once your application program has called `new_menu`, it should not change the array of item pointers until the menu is freed by `free_menu` or the item array is replaced by `set_menu_items`, 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 “Fetching and Changing Menu Items” on page 10-14.

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. You 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 Screen 10-2 for an example showing 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, you use function `free_menu`.

**SYNOPSIS**

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

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

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

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

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

For `E_POSTED,` see “Posting and Unposting Menus” on page 10-27.

## Manipulating Menu Attributes

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

### Fetching and Changing Menu Items

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

**SYNOPSIS**

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

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

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

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

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

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

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

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

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

As an example of `set_menu_items`, consider Screen 10-6 whose code changes the items associated with a previously created menu.

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

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

                              /* process menu with olditems */

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

**Screen 10-6. Changing the Items Associated with a Menu**

## Counting the Number of Menu Items

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

**SYNOPSIS**

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

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

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

```
/* check visibility of last menu item */
int at_bottom (m)
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 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, you use the functions `set_menu_win` or `set_menu_sub`. (These routines are treated “Associating Windows and Subwindows with Menus” on page 10-23.) Whether or not you choose a menu window, ETI calculates the minimum window (or subwindow) size for your menu.

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

- the size and number of items in a menu
- whether option `O_ROWMAJOR` is on
- whether option `O_SHOWDESC` is on
- the format, or maximum number of rows and columns on a displayed page of the menu
- the mark string for menu items

ETI knows the size and number of items in a menu as soon as you call `new_menu`, discussed above. By default, options `O_ROWMAJOR` and `O_SHOWDESC` are on. Option `O_ROW_MAJOR` ensures that the items are displayed in row major order — fanning out left to right, then top to bottom. How to change this and other menu options is discussed in “Setting and Fetching Menu Options” on page 10-47. Option `O_SHOWDESC` ensures that an item's description, if any, is displayed with the item's name.

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

**NOTE**

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

## Specifying the Menu Format

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

**SYNOPSIS**

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

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

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

The actual number of rows and columns displayed may be less than *maxrows* or *maxcols* depending on the number of items and whether the `O_ROWMAJOR` option is on. (Menu options are described in “Setting and Fetching Menu Options” on page 10-47.) 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 one. If either *maxrows* or *maxcols* equals zero 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 zero.

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 “ETI Menu Requests” on page 10-32.

As an example, consider the displays in Figure 10-1 and Figure 10-2. They portray menus consisting of five items. The numbers 0 through 4 signify menu items in the order in which they live in the item pointer array. Figure 10-1 shows the menu displayed with a format of maximum number of rows two, maximum number of columns two. To stipulate this format for menu `m`, you write

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

Using the formulas above, we see that `total_rows` is 3 and `total_cols` is 2 in all four cases displayed in the two figures. The first display in each figure shows the menu in row major format (`O_ROW_MAJOR` on), the second in column major format. The displayed number of rows and columns in Figure 10-1 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 three rows, two columns, you get one of the two displays in Figure 10-2. The enclosing block in each case indicates the items displayed at one time.

**Figure 10-1. Examples of Menu Format (2, 2)**

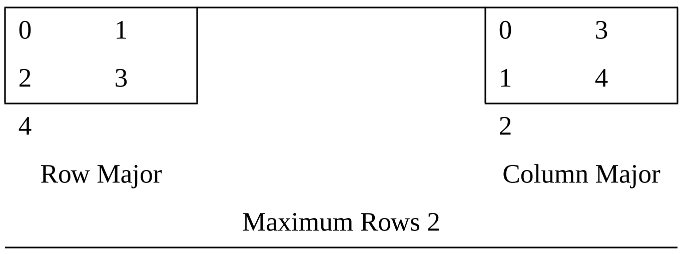
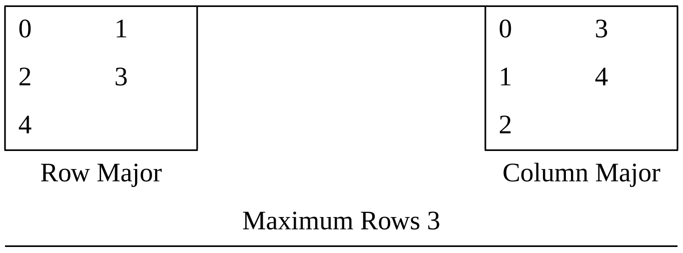


**Figure 10-2. Examples of Menu Format (3, 2)**



For a larger example, consider Figure 10-3. 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 six, the total number of columns to three, and the displayed number of rows to four. Calculation shows that changing the number of items in this example to 19 changes the number of rows to seven.

**Figure 10-3. Examples of Menu Format (4, 3)**

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 0 | 6 | 12 |
| 3 | 4 | 5 | 1 | 7 | 13 |
| 6 | 7 | 8 | 2 | 8 | 14 |
| 9 | 10 | 11 | 3 | 9 | 15 |
| 12 | 13 | 14 | 4 | 10 | 16 |
| 15 | 16 | 17 | 5 | 11 | 17 |
| | Row Major | | | Column Major | |

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

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

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

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

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

The function `set_menu_format` resets the value of `top_row` to 0. See “Fetching and Changing the Top Row” on page 10-42 for details.

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

## Changing Your Menu's Mark String

The mark string distinguishes

- selected items in a multi-valued menu
- the current item in a single-valued menu.

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

**SYNOPSIS**

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

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

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

**NOTE**

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

You can call `set_menu_mark` either before or after the menu is posted. (See “Posting and Unposting Menus” on page 10-27.) 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 or string is NULL |

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 `--->` you 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, you call `menu_mark` with NULL:

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

## Querying the Menu Dimensions

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

### SYNOPSIS

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

Because function `scale_menu` must return more than one value (namely, the minimum number of rows and columns for the menu) and C passes parameters “by value” only, the arguments of `scale_menu` are pointers. The pointer arguments *rows* and *cols* point to locations used to return the minimum number of rows and columns for displaying the menu.

**NOTE**

You should call `scale_menu` only after the menu's items have been connected to the menu using `new_menu` or `set_menu_items`.

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

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

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

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

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

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL menu pointer |
| `E_NOT_CONNECTED` | no connected items |

## Associating Windows and Subwindows with Menus

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

**SYNOPSIS**

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

WINDOW * menu_win (menu)
MENU * menu;

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

WINDOW * menu_sub (menu)
MENU * menu;
```

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

**NOTE**

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

If you do not want to use the system defaults, you may create a window and a subwindow for every menu. ETI automatically writes all output of the menu proper on the menu's subwindow. You may write additional output (such as borders, titles, and the like) on the menu's window. The relationship between ETI menu routines, your application program, a menu window, and a menu subwindow is illustrated in Figure 10-4.

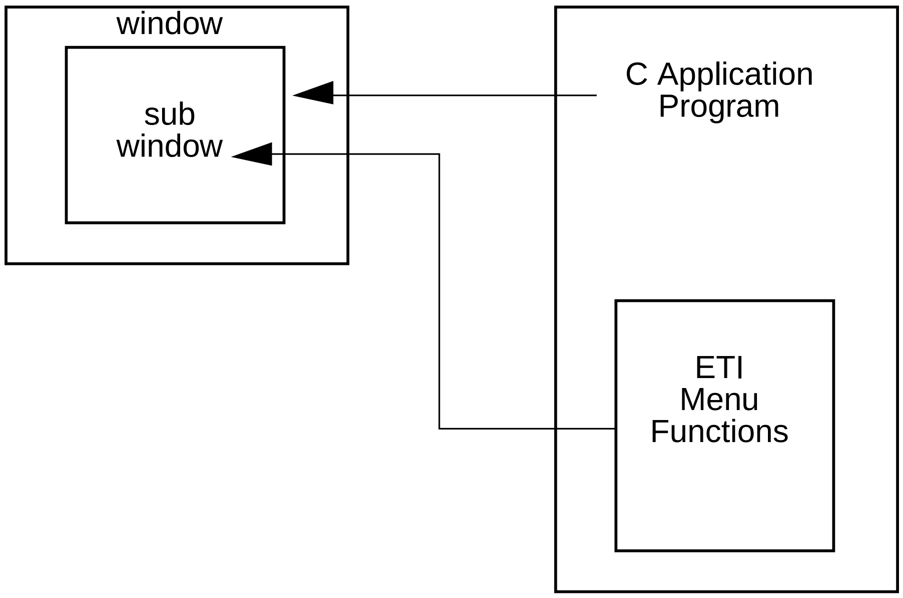


**Figure 10-4.  Menu Functions Write to Subwindow, Application to Window**

**NOTE**

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

Figure 10-7 shows how you can create and display a menu with a border of the default characters, `ACS_VLINE` and `ACS_HLINE`. (See the **`box`** command in the **`curses(3curses)`** manual pages.)

```
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 */
}
```

**Screen 10-7.  Creating a Menu with a Border**

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

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

```
WINDOW * dftwin;

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

changes the current default window to `dftwin`.

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

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_POSTED` | menu is posted |

## Fetching and Changing a Menu's Display Attributes

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

| | |
|---|---|
| foreground attribute | distinguishes the current item, if selectable, on all menus and selected items on multi-valued menus |
| background attribute | distinguishes selectable, but unselected, items on all menus |
| grey attribute | distinguishes unselectable items on multi-valued menus |
| pad character | the character that fills (pads) the space between a menu item's name and description |

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

**SYNOPSIS**

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

chtype menu_fore (menu)
MENU ** menu;

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

chtype menu_back (menu)
MENU ** menu;

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

chtype menu_grey (menu)
MENU ** menu;

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

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

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

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

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

Function `set_menu_grey` sets the **`curses`** attribute used to display nonselectable items in multi-valued menus. 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. As an example, to set the default grey attribute to `A_NORMAL`, you 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 (`.`), you 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` | non-printable pad character |

## Posting and Unposting Menus

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

### SYNOPSIS

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

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

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

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

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

Regarding E_NO_ROOM, recall from “Querying the Menu Dimensions” on page 10-22 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.

Screen 10-8 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. Function erase_menu unposts the menu and erases its associated window and subwindow.

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

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

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

    if (w = newwin (rows+2, cols+2, 0, 0)) {

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

        /* post menu */

    if (post_menu (m) != E_OK)
        error (“error return from post_menu”, NULL);
    else
        wrefresh (w);
}

static void erase_menu (m)     /* unpost and delete menu windows */
MENU * m;
{
    WINDOW * w = menu_win (m);
    WINDOW * s = menu_sub (m);

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

**Screen 10-8. Sample Routines Displaying and Erasing Menus**

Function `keypad` is called with a second argument of `1` to enable virtual keys `KEY_LL`, `KEY_LEFT`, and others to be properly interpreted in the routine `get_request` described in “Menu Driver Processing” on page 10-29. See the discussion of `keypad` in the **`curses(3curses)`** manual pages 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

- item navigation requests
- menu scrolling requests

- item selection requests
- pattern buffer requests

**SYNOPSIS**

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

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

To enable your application program to fetch the character for the menu driver, you 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 Screen 10-9, which shows the key virtualization routine `get_request`. Nearly all the values it returns are the ETI menu requests to be discussed in the sections below.

```
    /* define application commands */

#define QUIT(MAX_COMMAND + 1)

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

        ^Q       - end menu processing

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

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

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

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

        ^T       - toggle item     */

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

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

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

        case KEY_LEFT:        return  REQ_LEFT_ITEM;
        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;
}
```

**Screen 10-9. Sample Routine that Translates Keys into Menu Requests**

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

## ETI Menu Requests

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

A *menu page* is the collection of currently visible menu items,
that is, 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 Chapter 11.

### 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.

The `REQ_NEXT_ITEM` and `REQ_PREV_ITEM` requests are not cyclic. 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” on page 10-47.) 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 non-printable ASCII character that is received by the menu driver is assumed to be a menu request. On the other hand, each printable ASCII character that is received by the menu driver is entered into the pattern buffer. At the same time, the current item advances to the first matching item. If no matching item is found, the current item remains unchanged, the character is deleted from the pattern buffer, and the menu driver returns `E_NO_MATCH`.

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

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

Request `REQ_CLEAR_PATTERN` clears the pattern buffer entirely.

**NOTE**

Without request `REQ_CLEAR_PATTERN`, the pattern buffer is automatically cleared after each successful scrolling or item navigation operation. In other words, 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 occur in the example in the next section. Figure 10-5 diagrams this relationship between ETI key values, ETI menu requests, and your application program's menu requests.

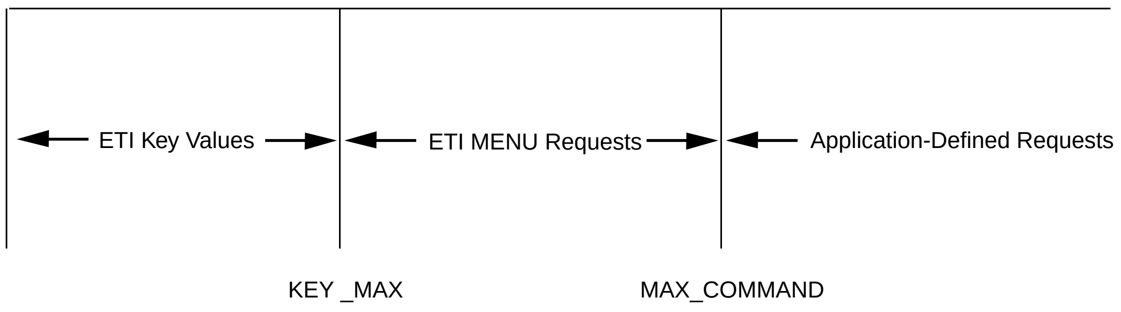
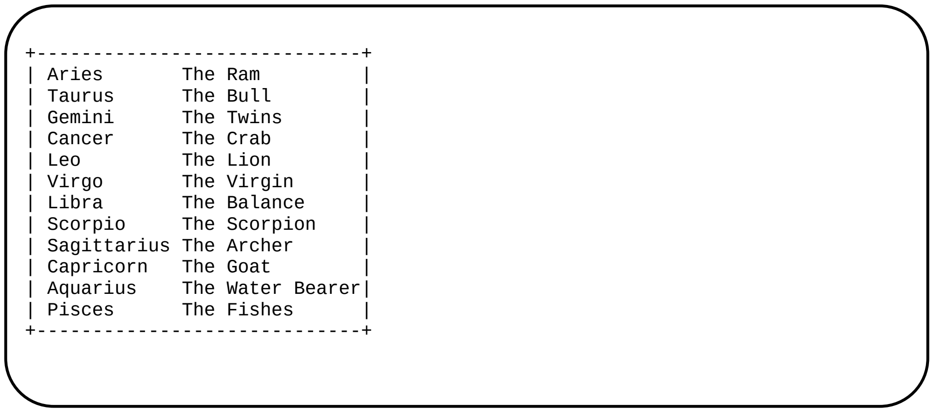


**Figure 10-5. Integer Ranges for ETI Key Values and MENU Requests**

## 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, that is, 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. Screen 10-10 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      |
+-----------------------------+
```

**Screen 10-10. Sample Menu Output (2)**

You have already seen much of the astrological sign program in previous examples. Its function `get_request`, for instance, appeared in Screen 10-9. Screen 10-11 shows its remaining routines.

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

        Omitted here are the key mapping defined by get_request
        in Screen 10-9; application-defined routines display_menu
        and erase_menu in Screen 10-8; and the curses initialization
        routine start_curses in section “ETI Low-level Interface to
        High-level Functions”   */

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

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


static int my_driver (m, c)/* handle application commands */
MENU * m;
int c;
{
    switch (c)
    {
        case QUIT:
            return TRUE;
            break;
    }
    beep ();/* signal error */
    return FALSE;
}

main (argc, argv)
int argc;
char * argv[];
{
    WINDOW *w;
    MENU *   m;
    ITEM **  i;
    ITEM **  make_items ();
    void     free_items ();
    int      c, done = FALSE;

    PGM = argv[0];
    start_curses ();

    if (! (m = new_menu (make_items ())))
        error (“error return from new_menu”, NULL);

    display_menu (m);

    /* interact with user */

    w = menu_win (m);

    while (! done)
    {
        switch (menu_driver (m, c = get_request (w)))
        {
            case E_OK:
                break;
            case E_UNKNOWN_COMMAND:
                done = my_driver (m, c);
                break;
            default:
                beep ();/* signal error */
                break;
        }
    }
```

**Screen 10-11.  Sample Program Calling the Menu Driver**

```
    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++);
}
```

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 and the signal prompts the user to re-enter the character. If the character passed is the QUIT character, `my_driver` returns TRUE. In turn, this sets `done` to TRUE, and the `while` loop is exited.

Finally, `main` erases the menu, terminates low-level ETI (**`curses`**), frees the menu and its items, and exits the program.

This example shows a typical design for calling the menu driver, but it is only one of several ways you can structure a menu application.

If the `menu_driver` recognizes and processes the input character argument, it returns `E_OK`. In the following error situations, the `menu_driver` returns the indicated value:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL menu |
| `E_BAD_STATE` | called from init/term routines |
| `E_NOT_POSTED` | menu is not posted |
| `E_UNKNOWN_COMMAND` | unknown command |
| `E_NO_MATCH` | item match failed |
| `E_REQUEST_DENIED` | recognized request failed |

**NOTE**

Because the menu driver calls the initialization and termination routines described in the next section, it may not be called from within them. Any attempt to do so returns `E_BAD_STATE`.

## Establishing Item and Menu Initialization and Termination Routines

Sometimes, you may want the menu driver to execute a specific routine during the change of an item or menu. The following functions let you do this easily.

**SYNOPSIS**

```
typedef void (*PTF_void) ();

int set_menu_init (menu, func)
MENU * menu;
PTF_void func;

PTF_void menu_init (menu)
MENU * menu;

int set_menu_term (menu, func)
MENU * menu;
PTF_void func;
PTF_void menu_term (menu)
MENU * menu;

int set_item_init (menu, func)
MENU * menu;
PTF_void func;

PTF_void item_init (menu)
MENU * menu;

int set_item_term (menu, func)
MENU * menu;
PTF_void func;

PTF_void item_term (menu)
MENU * menu;
```

The argument *func* is a pointer to the specific function you want executed by the menu driver. This application-defined function takes a menu pointer as an argument.

If you want your application to execute an application-defined function at one of the initialization or termination points listed below, you should call the appropriate `set_` routine at the start of your program. If you do not want a specific function executed in these cases, you may refrain from calling these routines altogether.

The following subsections 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, that is, 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 |

Screen 10-12 shows how you can use function `set_item_init` to implement a menu prompting feature as your end-user moves from item to item.

```
WINDOW * prompt_window;

void display_prompt (s)
char * s;
{
    WINDOW * w = prompt_window;

    werase (w);
    wmove (w, 0, 0);           /* move to window origin */
    waddstr (w, s);           /* write prompt in window */
    wrefresh (w);             /* display prompt */
}
void generate_prompt (m)
MENU * m;
{

    /* display the prompt string associated with the current item */

    char * s = item_userptr (current_item (m));
    display_prompt (s);
}
ITEM * items[NUMBER_OF_ITEMS + 1];

main ()
{
    MENU * m;

    for (i = 0; i < NUMBER_OF_ITEMS; ++i)
    {

        /* read in name and prompt strings here */

        items[i] = new_item (name, “”);
        set_item_userptr (items[i], prompt);
    }
    items[i] = (ITEM *) 0;

    m = new_menu (items);
    set_item_init (m, generate_prompt);  /* set initialization routine */
}
```

**Screen 10-12.  Using an Initialization Routine to Generate Item Prompts**

Function `set_item_init` arranges to call `generate_prompt` whenever the menu item changes. Function `generate_prompt` fetches the item user pointer associated with the current item and calls `display_prompt`, which displays the item prompt. Function `display_prompt` is a separate function to enable you to use it for other prompts as well.

## Fetching and Changing the Current Item

The current item is the item where your end-user is positioned on the screen. Unless it is invisible, this item is highlighted and the cursor rests on the item. To have your application program set or determine the current item, you use the following functions.

**SYNOPSIS**

```
int set_current_item (menu, item)
MENU * menu;
ITEM * item;

ITEM * current_item (menu)
MENU * menu;

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

Function `set_current_item` enables you to set the current item by passing an item pointer, while function `current_item` returns the pointer to the current item.

The function `item_index` takes an item pointer argument and returns the index to that item in the item pointer array. The value of this index ranges from 0 through N-1, where N is the total number of items connected to the menu.

Because the menu driver satisfies ETI-defined item navigation requests automatically, your application program need not call `set_current_item`, unless you want to implement additional item navigation requests for your application. You may, for instance, want a request to jump to a particular item or an item, say, two items down from the current one on the menu page.

When a menu is created by `new_menu` or the items associated with a menu are changed by `set_menu_items`, the current item is set to the first item of the menu.

As an example of `set_current_item`, the following function sets the current item of menu `m` to the first item of the menu:

```
int set_first_item (m) /* set current item to first
    item */
MENU *  m;
{
    ITEM ** i = menu_items (m);
    return set_current_item (m, i[0]);
}
```

As an example of `current_item`, the following routine checks if the first menu item is the current one:

```
int first_item (m)  /* check if current item is first
    item */
MENU * m;
{
    ITEM * i = current_item (m);
    return item_index (i) == 0;
}
```

If successful, function `set_current_item` returns `E_OK.` If an error occurs, function `set_current_item` returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL menu pointer or item not connected to menu |
| `E_BAD_STATE` | called from initialization or termination routines |

Function `current_item` returns (ITEM *) 0 if given a NULL menu pointer or there are no items connected to the menu.

Function `item_index` returns -1 if the item pointer is NULL or the item is not connected to a menu.

## Fetching and Changing the Top Row

Function `top_row` returns the number of the menu row currently displayed at the top of your end-user's menu. Function `set_top_row` sets the top of the menu to the named row, unless the row does not start a complete page of items. In this case, it returns `E_BAD_ARGUMENT`.

### SYNOPSIS

```
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” on page 10-18 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” on page 10-18 and “Setting and Fetching Menu Options” on page 10-47 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:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL menu pointer or index out of range |
| `E_BAD_STATE` | called from init/term routines |
| `E_NOT_CONNECTED` | no connected items |

Function `top_row` returns -1 if given a NULL menu pointer or no items are connected to the menu.

## Positioning the Menu Cursor

Some applications may need to move the menu's window cursor from the position required for continued processing by the ETI menu driver. To move the cursor back to where it belongs, you use function `pos_menu_cursor`.

### SYNOPSIS

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

If your application does not change the cursor position in the menu window, calling this function is unnecessary.

Your application might change the cursor position automatically because of prior calls to menu driver initialization routines such as `set_item_init`. Or it might do so because of explicit calls to application routines such as writing a prompt. Screen 10-13 illustrates this usage.

```
void generate_prompt (m)
MENU * m;
{

    /* display the prompt string associated with the current item */

    WINDOW * w = menu_win (m);
    char * s = item_userptr (current_item (m));
    box (w, 0, 0);
    wmove (w, 0, 0);
    waddstr (w, s);
    pos_menu_cursor (m);
}
```

**Screen 10-13. Returning Cursor to Its Correct Position for Menu Driver Processing**

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

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL menu pointer |
| `E_NOT_POSTED` | menu is not posted |

## Changing and Fetching the Pattern Buffer

Remember that the pattern buffer is used to make the first item that matches the pattern be the current item. In general, to match the current menu item, your application program inserts characters into the pattern buffer that have been passed to the menu driver from the user's data entry. As an alternative, you can insert characters into the pattern buffer with the function `set_menu_pattern`.

### SYNOPSIS

```
int set_menu_pattern (menu, pattern)
MENU * menu;
char * pattern;

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

Function `set_menu_pattern` first clears the pattern buffer and then adds the characters in `pattern` to the buffer until `pattern` is exhausted. The function next tries to find the first item that matches the `pattern`. If it does not find a complete match, the pattern buffer is cleared and the current item does not change. If `pattern` is the null string (“”), the pattern buffer is simply cleared. The pattern buffer is automatically cleared whenever

- 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, that is, (char *) 0.

To determine if your user has entered data that matches an item, you might write a routine that uses `set_menu_pattern`, as follows:

```
int find_match (m, newpattern)
    /* returns TRUE or FALSE */
MENU * m;
char * newpattern;
{
    return set_menu_pattern(m, newpattern) == E_OK;
}
```

If the `newpattern` matches a menu item, function `set_menu_pattern` returns `E_OK` and hence `find_match` returns TRUE. In addition, `find_match` advances the current item to the matching item.

## Manipulating the Menu User Pointer

As it does for panels and forms, ETI provides user pointers for each menu. You can use these pointers to reference menu messages, titles, and the like.

### SYNOPSIS

```
int set_menu_userptr (menu, userptr)
MENU * menu;
char * userptr;

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

By default, the menu user pointer (what `menu_userptr` returns) is NULL.

If successful, `set_menu_userptr` returns `E_OK.` If an error occurs, it returns the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |

The code in Screen 10-14 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 Screen 10-8.

```
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);
}
```

**Screen 10-14. Example Setting and Using a Menu User Pointer**

If function `set_menu_userptr` is passed a NULL menu pointer, like all ETI functions, it assigns a new current default menu user pointer. In the following, the new default is the string `Default Menu Title`.

```
MENU * m;

char * userprtr = “Default Menu Title”;

set_menu_userptr( (MENU *) 0, userptr);
        /* sets new default userptr */
```

## Setting and Fetching Menu Options

ETI provides several menu options, some of which we have already met. Two functions manipulate options: one sets them, the other returns their settings.

**SYNOPSIS**

```
int set_menu_opts (menu, opts)
MENU * menu;
OPTIONS opts;

OPTIONS menu_opts (menu)
MENU * menu;

options:
    O_ONEVALUE
    O_SHOWDESC
    O_ROWMAJOR
    O_IGNORECASE
    O_SHOWMATCH
```

Besides turning the named options on, function `set_menu_opts` turns off all other menu options. By default, all menu options are on.

The menu options and their effects are as follows:

| | |
|---|---|
| `O_ONEVALUE` | determines whether the menu is a single-valued or multi-valued. In general, menus are single-valued and this option is on. Recall that upon exit from single-valued menus, your application queries the current item to ascertain the item selected. Turning off this option signifies a multi-valued menu. One way to select several items is to use the `REQ_TOGGLE_ITEM` request, another is to call `set_item_value`. (See “Multi-valued Menu Selection Request” on page 10-33 and “Manipulating an Item's Select Value in a Multi-valued Menu” on page 10-7.) 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 displayed first 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” on page 10-18 for more 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. |

Like all ETI options, menu `OPTIONS` are Boolean values, so you use Boolean operators to turn them on or off with functions `set_menu_opts` and `menu_opts`. For example, to turn off option `O_SHOWDESC` for menu `m0` and turn on the same option for menu `m1`, you can write:

```
MENU * m0, * m1;

set_menu_opts (m0, menu_opts (m0) & ~O_SHOWDESC);
        /* turn option off */
set_menu_opts (m1, menu_opts (m1) |  O_SHOWDESC);
        /* turn option on  */
```

ETI provides two alternative functions for turning options on and off for a given menu.

**SYNOPSIS**

```
int menu_opts_on (menu, opts)
MENU * menu;
OPTIONS opts;

int menu_opts_off (menu, opts)
MENU * menu;
OPTIONS opts;
```

Unlike function `set_menu_opts`, these functions do not affect options that are unmentioned in their second argument. In addition, if you want to change one option, you need not apply Boolean operators or use `menu_opts`.

As an 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:

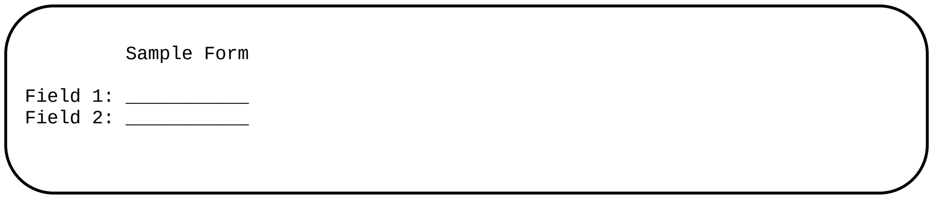
| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_POSTED` | menu is posted |

# 11 Forms

# 11 Forms

## Introduction

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. Screen 11-1 displays a simple form with five fields including two for data entry.



```
            Sample Form

Field 1: ___________
Field 2: ___________
```

**Screen 11-1. Sample Form Display**

## Compiling and Linking Form Programs

To use the form routines, you specify

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

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

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

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

## Overview: Writing Form Programs in ETI

This section introduces the basic ETI form terminology, lists the steps in a typical form application, and reviews the sample program that produced the output of Screen 11-1

## Some Important Form Terminology

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

| | |
|---|---|
| field | an *m* x *n* block of form character positions 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 |
| dynamic field | a field whose buffer grows beyond its original size if more data is entered into the field than the original buffer will hold. |
| form | a collection of one or more pages of fields |
| connecting fields to a form | associating an array of field pointers with a form |
| page | a logical subdivision of a form usually occupying one screen |
| posting a form | writing a form on its associated subwindow |
| unposting a form | erasing a form from its associated subwindow |
| freeing a form | deallocating the memory for a form and, as a by-product, disconnecting the previously associated array of field pointers from the form |
| freeing a field | deallocating the memory for a field |
| NULL | generic term for a null pointer cast to the type of the particular object (field, form, and so on) |

## What a Typical Form Application Program Does

In general, a form application program will

- initialize low-level ETI (**`curses`**)
- create the fields for the form
- create the form
- post the form
- refresh the screen
- process end user form requests
- unpost the form
- free the form

- free the fields
- terminate low-level ETI (**`curses`**)

## A Sample Form Application Program

Screen 11-2 shows the ETI program necessary for producing the form in Screen 11-1.

```
#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;
```

**Screen 11-2. Code to Produce a Simple Form**

```
/*
    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(3C)`** for details.

Next, there are two programmer-defined functions `make_label` and `make_field`, which we will discuss in a moment. Consider procedure `main`. It declares three objects:

- `form`, a pointer to a form
- `f[6]`, an array of field pointers
- `i`, an index variable, initialized to 0

The first five functions initialize low-level ETI (**`curses`**) for high-level ETI form functions. Function `initscr` initializes the screen, `nonl` ensures that a carriage return on using `wgetch` will not automatically generate a newline, `raw` passes input characters without interpretation to your program, `noecho` disables echoing of your user's input (the form functions provide echoing where appropriate), and `wclear(stdscr)` clears the standard screen.

The statements that create the form's fields and labels in this example make calls to the programmer-defined functions `make_label` and `make_field`. You can do without these programmer-defined functions, but you may find them convenient. Both of them use the ETI function `new_field`. They take three arguments, which correspond to three of the six arguments of `new_field`.

The first argument of `new_field` is the number of rows of the field. In this example, it is always one. The last two arguments are often 0 as they are here; they will be explained in the next section. The second argument of `new_field` is the number of columns in the field. This number is determined from the third parameter in `main`'s calls to `make_label` and `make_field`. For the label fields, the calls to `make_label` pass the string that is to constitute the field so that `strlen` can be used to count the length or number of columns of the string. For the fields to be edited by the end-user (had this example permitted entering data into the fields), calls to `make_field` simply pass the number of columns directly.

The third and fourth arguments to `new_field` correspond to the first and second arguments to `make_label` and `make_field`. They are the starting position (`firstrow`, `firstcol`) of the label or field in the form subwindow. (In this example, the default subwindow `stdscr` is used.) The last assignment to `f[5]` terminates the array with the NULL field pointer.

Once the function `make_label` creates the field for the label, it places the label in the field using function `set_field_buffer`. The second argument to this function is 0 because the value of a field is stored in buffer 0. Finally, function `make_label` calls `set_field_opts`, which turns off the `O_ACTIVE` option for the field. This means that the field is ignored during form driver processing.

On the other hand, once the function `make_field` creates the field proper, it sets the field's background attribute to `A_UNDERLINE`. This has the effect of underlining the field so that it is visible.

After you create the fields for a form, you create the form itself using `new_form`. This function takes the pointer to the array of field pointers and connects the fields to the form. The pointer returned is stored in variable `form` — it will be passed to subsequent form manipulation routines. To display the form, function `post_form` posts it on the default subwindow `stdscr`, while `wrefresh(stdscr)` actually displays this subwindow on the terminal screen. The display remains for 5 seconds, as determined by `sleep`.

At this point, most forms would accept and process user input. To illustrate a very simple form, this program does not accept user input.

To erase the form, you first unpost it using `unpost_form`. This erases it from the form subwindow. The call to `wrefresh` actually erases the form from the display screen. Function `free_form` disconnects the form from its array of field pointers `f`.

The `while` loop, starting with the first field in the field pointer array, frees each field referenced in the array. The effect is to deallocate the space for each field.

We have met the last two lines of the program before. Function `endwin` terminates low-level ETI, while **`exit(0)`** terminates the program.

There are many ETI form routines not listed in Screen 11-2. These routines enable you to tailor your form programs to suit local needs and preferences. The following sections explain how to use all ETI form routines. Each routine is illustrated with one or more code fragments. Many of these are drawn from two larger form application programs listed at the end of the chapter. By reviewing the code fragments, you will come to understand the larger programs.

## Creating and Freeing Fields

To create a form, you must first create its fields. The following functions enable you to create fields and later free them.

**SYNOPSIS**

```
FIELD * new_field (rows, cols, firstrow, firstcol,  nrow, nbuf)
int rows,  cols,  firstrow, firstcol,  nrow, nbuf;

FIELD * dup_field (field,  firstrow,  firstcol)
FIELD * field;
int firstrow,  firstcol;

FIELD * link_field (field, firstrow, firstcol)
FIELD * field;
int firstrow, firstcol;

int free_field (field)
FIELD * field;
```

Unlike menu items which always occupy one row, the fields on a form may contain one or more rows. Function `new_field` creates and initializes a new field that is *rows* by *cols* large and starts at point (*firstrow*, *firstcol*) relative to the origin of the form subwindow. All current system defaults are assigned to the new field when it is created using `new_field`.

Variable *nrow* is the number of offscreen rows allocated for this field. Offscreen rows enable your program to display only part of a field at a given moment and let the user scroll through the rest. A zero value means that the entire field is always displayed, while a nonzero value means that the field is scrollable. A field can be created with *nrow* set to zero and allowed to grow and scroll if the field is made dynamic. See “Dynamically Growable Fields” on page 11-9 for more detail.

Variable *nbuf* is the number of additional buffers allocated for this field. You can use it to support default field values, undo operations, or other similar operations requiring one or more auxiliary field buffers.

Variables *rows* and *cols* must be greater than zero, while *firstrow*, *firstcol*, *nrow*, and *nbuf* must be greater than or equal to zero.

Each field buffer is ((*rows* + *nrow*) * *cols* + 1) characters large. (The extra character position holds the NULL terminator.) All fields have one buffer (namely, field buffer 0) that maintains the field's value. This buffer reflects any changes your end-user may make to the field. See “Setting and Reading Field Buffers” on page 11-21 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 Screen 11-2 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 “Creating and Freeing Forms” on page 11-29 and “Manipulating Field Options” on page 11-26 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 “Manipulating Field Options” on page 11-26.

**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 “Creating and Freeing Forms” on page 11-29, 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 and Fetching the Fields on an
    Existing Form” */

FIELD ** f = form_fields (form);

free_form (form);          /* free form */

while (*f)
    free_field (*f++);
        /* free each field and increment pointer */
```

Notice that you free the form before its fields.

If successful, function `free_field` returns `E_OK`. If not, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL field pointer |
| `E_CONNECTED` | connected field |

Remember that the field pointer returned by `new_field`, `dup_field`, or `link_field` is passed to all field routines that record or examine the field's attributes. As with menu items, once a form field is freed, it must not be used again. Because the freed field pointer does not point to a genuine field, undefined results occur.

## Manipulating Field Attributes

Recall that an attribute is any feature whose value can be set or read by an appropriate ETI function. A field attribute is a feature of a field whose value can be set or read by an appropriate ETI function. Field attributes include the field size and location.

### Obtaining Field Size and Location Information

This function enables you to determine the defining characteristics of a field — its size, position, number of offscreen rows, and number of associated buffers.

#### SYNOPSIS

`int field_info (`*field*`,` *rows*`,` *cols*`,` *firstrow*`,` *firstcol*`,` *nrow*`,` *nbuf*`)`
`FIELD *` *field*`;`
`int *` *rows*`, *` *cols*`, *` *firstrow*`, *` *firstcol*`, *` *nrow*`, *` *nbuf*`;`

Because function `field_info` must return more than a single value and C passes arguments to functions “by 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 off-screen rows, and the number of additional buffers.

As an example, consider how you might use `field_info` to determine a field's buffer size. You fetch the field's number of onscreen and offscreen rows and number of columns, and do the arithmetic, thus:

```
int bufsize (f)
FIELD * f;
{
    int rows, cols, firstrow, firstcol, offrow, nbuf;

    field_info (f, &rows, &cols, &firstrow, &firstcol,
        &offrow, &nbuf);

    /* add up size of field and its terminator */

    return (rows + offrow) * cols + 1;
}
```

Note the use of the address operator `&` to pass `field_info` the requisite pointers to the locations used to return the requested field information.

If successful, function `field_info` returns `E_OK`. If not, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL field pointer |

## Dynamically Growable Fields

A dynamically growable field within a form will allow a user to add more data to a field than was specified when the field was originally created. Recall, when a field is created, a buffer is allocated based on the size of the field. With dynamically growable fields, if a user enters more data than the original buffer can hold, the buffer will grow as the user enters more data into the field. The application developer can specify the maximum growth of a field or allow a field to grow without bound.

A field can be made dynamically growable by turning off the `O_STATIC` field option. See “Manipulating Field Options” on page 11-26 for more information on changing field options.

Recall the library routine `new_field`; a new field created with *rows* set to one and *nrow* set to zero will be defined to be a one line field. A new field created with *rows* + *nrow* greater than one will be defined to be a multi-line field.

A one line field with `O_STATIC` turned off will contain a single fixed row, but the number of columns can increase if the user enters more data than the initial field will hold. The number of columns displayed will remain fixed and the additional data will scroll horizontally.

A multi-line field with `O_STATIC` turned off will contain a fixed number of columns, but the number of rows can increase if the user enters more data than the initial field will hold.

The number of rows displayed will remain fixed and the additional data will scroll vertically.

It may be desirable to allow a field to grow, but within bounds. The following function can be used to limit the growth of a dynamic field either horizontally or vertically.

### SYNOPSIS

```
int set_max_field(field, max_growth)
FIELD *field;
int max_growth;
```

If *field* is a horizontally growable one line field, its growth will be limited to *max_growth* columns. If *field* is a vertically growable field, its growth will be limited to *max_growth* rows. To remove any growth limit, call `set_max_field` with *max_growth* set to zero. To query the current maximum, if specified, see `dynamic_field_info` below.

If successful this procedure will return `E_OK`, otherwise the following is returned:

| | |
|---|---|
| `E_BAD_ARGUMENT` | NULL field pointer or field size is already greater than `max_growth` or `max_growth` is less than zero. |

This procedure will work regardless of the setting of the `O_STATIC` option.

In order to allow the user to query the current size of the buffer, the following function is provided.

### SYNOPSIS

```
int dynamic_field_info(field, drows, dcols, max)
FIELD *field;
int   *drows, *dcols, *max;
```

If successful this procedure will return `E_OK`, and *drows* and *dcols* will contain the actual number of rows and columns of `field`. If a maximum growth has been specified (see `set_max_field` above) for *field*, *max* will contain the specified growth limit, otherwise *max* will contain zero.

If *field* is NULL, *drows*, *dcols*, and *max* are unchanged and the following is returned:

| | |
|---|---|
| `E_BAD_ARGUMENT` | NULL field pointer |

This procedure will work regardless of the setting of the `O_STATIC` option.

Making a field dynamic by turning off the `O_STATIC` option will affect the field in the following ways:

1. If parameter *nbuf* in the original `new_field` library call is greater than zero, all additional buffers will grow simultaneously with buffer 0. Recall, buffer 0 is used by the system to store data entered by the user, *nbuf* can be used to request the allocation of additional buffers available to the application. The field buffers will grow in chunks of size *buf_size* `= ((`*rows* `+` *nrow*`) *` *cols*`)`, the size of the original buffer minus one.

If a field is dynamic, the remainder of the forms library is affected in the following way.

1. The field option `O_AUTOSKIP` will be ignored if the option `O_STATIC` is off and there is no maximum growth specified for the field. Currently, `O_AUTOSKIP` generates an automatic `REQ_NEXT_FIELD` form driver request when the user types in the last character position of a field. On a growable field with no maximum growth specified, there is no “last” character position. If a maximum growth is specified, the `O_AUTOSKIP` option will work as normal if the field has grown to its maximum size.

2. The field justification will be ignored if the option `O_STATIC` is off. Currently, `set_field_just` can be used to `JUSTIFY_LEFT`, `JUSTIFY_RIGHT`, `JUSTIFY_CENTER` the contents of a one line field. A growable one line field will, by definition, grow and scroll horizontally and may contain more data than can be justified. The return from `field_just` will be unchanged.

3. The overloaded form driver request `REQ_NEW_LINE` will operate the same way regardless of the `O_NL_OVERLOAD` form option if the field option `O_STATIC` is off and there is no maximum growth specified for the field. Currently, if the form option `O_NL_OVERLOAD` is on, `REQ_NEW_LINE` implicitly generates a `REQ_NEXT_FIELD` if called from the last line of a field. If a field can grow without bound, there is no last line, so `REQ_NEW_LINE` will never implicitly generate a `REQ_NEXT_FIELD`. If a maximum growth limit is specified and the `O_NL_OVERLOAD` form option is on, `REQ_NEW_LINE` will only implicitly generate `REQ_NEXT_FIELD` if the field has grown to its maximum size and the user is on the last line.

4. The library call `dup_field` will work as described in “Creating and Freeing Fields” on page 11-6; it will duplicate the field, including the current buffer size and contents of the field being duplicated. Any specified maximum growth will also be duplicated.

5. The library call `link_field` will work as described in the section “Creating and Freeing Fields” on page 11-6; it will duplicate all field attributes and share buffers with the field being linked. If the `O_STATIC` field option is subsequently changed by a field sharing buffers, how the system reacts to an attempt to enter more data into the field than the buffer will currently hold will depend on the setting of the option in the current field.

6. The library call `field_info` will work as described in “Obtaining Field Size and Location Information” on page 11-8; the variable `nrow` will contain the value of the original call to `new_field`. The user should use `dynamic_field_info`, described above, to query the current size of the buffer.

## Moving a Field

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

### SYNOPSIS

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

`int` *firstrow*;
`int` *firstcol*;

Screen 11-3 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.

```
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;
    }
}
```

**Screen 11-3. Example Shifting All Form Fields a Given Number of Rows**

See “Obtaining Field Size and Location Information” on page 11-8 for more on `field_info` used in this example.

If successful, function `move_field` returns `E_OK`. If not, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL field or firstrow/firstcol < 0 |
| `E_CONNECTED` | connected field |

## Changing the Current Default Values for Field Attributes

ETI establishes initial current default values for field attributes. During field initialization, every field attribute is assigned the current default value for the attribute. As you can with menu functions, you can change or retrieve the current default attribute values by calling the appropriate function with a NULL field pointer. After the current default changes, every field created using `new_field` will have the new default value.

**NOTE**

Fields previously created do not have their attributes changed by changing the current system default.

Several of the following sections show how to change the default values for various field attributes.

## Setting the Field Type to Ensure Validation

Every field is created with the current default field type. The initial ETI default field type is a no_validation field. Any data may occupy it. (This default can be changed as described below.) To change a field's type from the default, ETI provides the following functions for manipulating a field's (data) type.

### SYNOPSIS

```
int set_field_type (field, type, [arg_1, arg_2, ...])
FIELD * field;
FIELDTYPE * type;

FIELDTYPE * field_type (field)
FIELD * field;

char * field_arg (field)
FIELD * field;
```

The function `set_field_type` takes a `FIELDTYPE` pointer and a variable number of arguments depending on the field type. The field type ensures that the field is validated as your end-user enters characters into the field or attempts to leave it.

The form driver (described later in “Form Driver Processing” on page 11-40) 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“Form Driver Processing” on page 11-40.

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 “Supporting Programmer-defined Field Types” on page 11-68.

If the function `set_field_type` is not applied to a field, the field type is the 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 in “Creating and Manipulating Programmer-defined Field Types” on page 11-65. The following sections describe all ETI generic field types.

## TYPE_ALPHA

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

### SYNOPSIS

```
set_field_type (field, TYPE_ALPHA, width);
int width; /* minimum token width */
```

`TYPE_ALPHA` takes one additional argument, the minimum width specification of the field. Note that when you previously create a field with function `new_field`, your *cols* argument is the maximum width specification of the field. With `TYPE_ALPHA` (and `TYPE_ALNUM` as well), your specification *width* must be less than or equal to *cols*. If not, the form driver cannot validate the field.

**NOTE**

`TYPE_ALPHA` does not allow blanks or other special characters.

To set a `middlename` field, for instance, to `TYPE_ALPHA` with a minimum of 0 characters (in effect, to make the end-user's completing the field optional), you can write

```
FIELD * middlename;

set_field_type (middlename, TYPE_ALPHA, 0);
```

## TYPE_ALNUM

This type restricts the set field to alphanumeric data, alphabetic characters (upper- or lower-case) and digits.

### SYNOPSIS

```
set_field_type (field, TYPE_ALNUM, width);
int width; /* minimum token width */
```

Like TYPE_ALPHA, TYPE_ALNUM takes one additional argument, the field's minimum width specification.

**NOTE**

Like TYPE_ALPHA, TYPE_ALNUM does not allow blanks or other special characters.

To set a field, say `partnumber`, to receive alphanumeric data at least eight characters wide, you write

```
FIELD * partnumber;

set_field_type (partnumber, TYPE_ALNUM, 8);
```

## TYPE_ENUM

This field type enables you to restrict the valid data for a field to a set of enumerated values. The type takes three arguments beyond the minimum two that `set_field_type` requires.

### SYNOPSIS

```
set_field_type (field, TYPE_ENUM, keyword_list, checkcase,
   checkunique);
char ** keyword_list; /* list of acceptable values */
int checkcase;        /* check character case   */
int checkunique;     /* check for unique match    */
```

The argument *keyword_list* is a NULL-terminated array of pointers to character strings that are the acceptable enumeration values. Argument *checkcase* is a Boolean flag that indicates whether upper- or lower-case is significant during match operations. Finally, *checkunique* is a Boolean flag indicating whether a unique match is required. If it is off and your end-user enters only part of an acceptable value, the validation procedure completes the field value automatically with the first matching value in the type. If it is on, the validation procedure completes the field value automatically only when enough characters have been entered to make a unique match.

To create a field, say `response`, with valid responses of `yes` (`y`) or `no` (`n`) in upper- or lower-case, you write:

```
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 Screen 11-4 which sets the `TYPE_ENUM` of field `color` to a list of colors.

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

set_field_type (color, TYPE_ENUM, colors, FALSE, TRUE);
```

**Screen 11-4. Setting a Field to TYPE_ENUM of Colors**

Setting the field to TRUE requires the user to enter the seventh character of the color name in certain cases (`Light Blue` and `Light Gray`) before a unique match is made.

## TYPE_INTEGER

This type enables you to restrict the data in a field to integers.

### SYNOPSIS

```
set_field_type (field, TYPE_INTEGER, precision, vmin, vmax);
int precision;   /* width for left padding with 0's */
long vmin;       /* minimum acceptable value */
long vmax;       /* maximum acceptable value */
```

`TYPE_INTEGER` takes three additional arguments: a precision specification, a minimum acceptable value, and a maximum acceptable value.

As your end-user enters characters, they are checked for validity. A `TYPE_INTEGER` value is valid if it consists of an optional minus sign followed by some number of digits. As the end-user tries to leave the field, the range check is applied.

**NOTE**

If, contrary to possibility, the maximum value *vmax* is less than or equal to the minimum value *vmin*, the range check is ignored — any integer that fits in the field is valid.

If the range check is passed, the integer is padded on the left with zeros to the precision specification. For instance, if the current value were 18, a precision of 3 would display

```
018
```

whereas a precision of 4 would display

```
0018
```

For more on ETI's handling of precision, see the manual page **`printf(3S)`**.

As an example of how to use `set_field_type` with `TYPE_INTEGER`, the following might represent a month, padded to two digits:

```
FIELD * month;

set_field_type (month, TYPE_INTEGER, 2, 1L, 12L);
/* displays single digit months with leading 0 */
```

Note the requirement that the minimum and maximum values be converted to type `long` with the `L`.

## TYPE_NUMERIC

This type restricts the data for the set field to decimal numbers.

### SYNOPSIS

```
set_field_type (field, TYPE_NUMERIC, precision, vmin, vmax);
int precision;   /* digits to right of the decimal point */
double vmin;   /* minimum acceptable value */
double vmax;   /* maximum acceptable value */
```

`TYPE_NUMERIC` takes three additional arguments: a precision specification, a minimum acceptable value, and a maximum acceptable value.

As your end-user enters characters, they are checked for validity as decimal numbers. A `TYPE_NUMERIC` value is valid if it consists of an optional minus sign, some number of digits, a decimal point, and some additional digits.

The precision is not used in validation; it is used only in determining the output format. See **`printf(3S)`** for more on precision. As the end-user tries to leave the field, the range check is applied.

As with `TYPE_INTEGER`, if the maximum value is less than or equal to the minimum value, the range check is ignored.

For instance, to set a maximum value of $100.00 for a monetary field `amount`, you write:

```
FIELD * amount;

set_field_type (amount, TYPE_NUMERIC, 2, 0.00, 100.00);
```

## TYPE_REGEXP

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

**SYNOPSIS**

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

TYPE_REGEXP takes one additional argument, the regular expression. See **regcmp(3G)** for regular expression details.

Consider, for example, how you might create a field that represents a part number with an upper- or lower-case letter followed by exactly 4 digits:

```
FIELD * partnumber;

set_field_type (partnumber, TYPE_REGEXP,
   “^[A-Za-z][0-9]{4}$”);
```

Note that this example assumes the field is five characters wide. If not, you may want to change the pattern to accept blanks on either side, thus:

```
FIELD * partnumber;

set_field_type (partnumber, TYPE_REGEXP,
   “^ *[A-Za-z][0-9]{4} *$”);
```

## Justifying Data in a Field

Unlike menu items, which always occupy one line, form fields may occupy one or more lines (rows). Fields that occupy one line may be justified left, right, center, or not at all.

**SYNOPSIS**

```
int set_field_just (field, justification)
FIELD * field;
int justification;

int field_just (field)
FIELD * field;
```

Fields that occupy more than one line are not justified because the data entered typically extends into subsequent lines. Justification is also ignored on a one line field if the O_STATIC option is off or the field was dynamic and has grown beyond its original size. See “Dynamically Growable Fields” on page 11-9 for more detail.

Field contents justification is not allowed for non-editable fields. However, if the field was already justified before making it, it will remain justified.

Setting the number of field columns (*cols*) and the minimum width or precision does not always determine where the data fits in the field — there may be excess character space before or after the data. Function set_field_just lets you justify data in one of the following ways:

| | |
|---|---|
| NO_JUSTIFICATION | no justification processing (initial default) |

| | |
|---|---|
| `JUSTIFY_LEFT` | left justify value in field |
| `JUSTIFY_RIGHT` | right justify value in field |
| `JUSTIFY_CENTER` | center value in the field |

No matter what the justification, fields are automatically left justified as your end-user enters data and edits the field. Once field validation occurs upon the user's request to leave the field, ETI justifies the field as specified.

For instance, to left justify a name field and right justify an amount field, you can write:

```
FIELD * name, * amount;

set_field_just (name, JUSTIFY_LEFT);
                /* left justify a field */

set_field_just (amount, JUSTIFY_RIGHT);
                /* right justify a field */
```

If successful, `set_field_just` returns `E_OK`. If not, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | bad justification |
| `E_REQUEST_DENIED` | justification request denied |

As with most other ETI functions, if one of these functions is passed a NULL field pointer, it assigns or fetches the system default. For instance, to change the system default from no justification to centering the value in its field, you write

```
set_field_just( (FIELD *) 0, JUSTIFY_CENTER);
                /* set new default */
```

## Setting the Field Foreground, Background, and Pad Character

The following functions enable you to set and read the pad character and the low-level ETI (**`curses`**) attributes associated with your field's foreground and background. The foreground attribute applies only to those field characters that represent data proper, while the background attribute applies to the entire field.

### SYNOPSIS

```
int set_field_fore (field, attr)
FIELD * field;
chtype attr;

chtype field_fore (field)
FIELD * field;
```

```
int set_field_back (field, attr)
FIELD * field;
chtype attr;

chtype field_back (field)
FIELD * field;

int set_field_pad (field, pad)
FIELD * field;
int pad;

int field_pad (field)
FIELD * field;
```

The initial default for both the foreground and background are `A_NORMAL`. (See the section on attribute descriptions earlier in this guide or the **`curses(3curses)`** pages 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` | unprintable pad character |

As usual, you can change or access the ETI defaults. To change the default background to `A_UNDERLINE`, you write:

```
set_field_back ((FIELD *) 0, A_UNDERLINE);
```

## Some Helpful Features of Fields

ETI provides special features that promote development of a wide range of form applications. These include field buffers, field status flags, and field user pointers.

### Setting and Reading Field Buffers

Recall that you set the number of additional buffers associated with a field upon its creation with `new_field`. Buffer 0 holds the value of the field. The following functions let you store values in the buffers and later read them.

**SYNOPSIS**

```
int set_field_buffer (field, buffer, value)
FIELD * field;
int buffer;
char * value;

char * field_buffer (field, buffer)
FIELD * field;
int buffer;
```

The parameter *buffer* should range from 0 through *nbuf*, where `nbuf` is the number of additional buffers in the `new_field` call. All buffers besides 0 may be used to suit your application.

If *field* in `set_field_buffer` is a dynamic field and the length of *value* is greater than the current buffer size, the buffer will expand, up to the specified maximum, if any, to accommodate *value*. See “Dynamically Growable Fields” on page 11-9 for more detail on dynamic fields and setting a maximum growth. If the field is not dynamic or the length of *value* is greater than any specified maximum field size, then *value* may be truncated.

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 “Manipulating the Current
           Field” below */

        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 “Creating a Field Type with Validation Functions” on page 11-66, “Establishing Field and Form Initialization and Termination Routines” on page 11-53, and “Field Validation Requests” on page 11-47 for details on these routines.

## Setting and Reading the Field Status

Every field has an associated status flag that is set whenever the field's value (field buffer 0) changes. The following functions enable you to set and access this flag.

**SYNOPSIS**

```
int set_field_status (field, status)
FIELD * field;
int status;

int field_status (field)
FIELD * field;
```

The field status is TRUE if set or FALSE if cleared. By default, the field status is FALSE when the field is created.

These routines promote increased efficiency where processing need occur only if a field has been changed since some previous state. Two examples are undo operations and database updates. Function `update` in Screen 11-5 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).

```
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;
    }
}
```

**Screen 11-5. Using the Field Status to Update a Database**

If successful, `set_field_status` returns `E_OK`. If not, it returns the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |

The initial ETI default field status is clear. As always, you can change the default by passing `set_field_status` a NULL field pointer.

Like the function `field_buffer`, function `field_status` always returns the correct value if the field is not current. However, if the field is current, the function is sometimes inaccurate because the status flag is not set immediately. You may rest assured that `field_status` is accurate on the current field if

- it is called from the field check validation routine, if any
- it is called from the form or field initialization or termination routines, if any
- it is called just after a `REQ_VALIDATION` request to the form driver

See “Creating a Field Type with Validation Functions” on page 11-66, “Establishing Field and Form Initialization and Termination Routines” on page 11-53, and “Field Validation Requests” on page 11-47 for details on these routines.

## Setting and Fetching the Field User Pointer

As it does with panels and menus, ETI provides functions to manipulate an arbitrary pointer convenient for field data such as title strings, help messages, and the like.

### SYNOPSIS

```
int set_field_userptr (field, userptr)
FIELD * field;
char * userptr;

char * field_userptr (field)
FIELD * field;
```

You can connect an application-defined structure to the field using this pointer. By default, the field user pointer is NULL.

Screen 11-6 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_field` | searches the names associated with all fields on the form to determine whether any of them match an arbitrary name passed to it |

```
#define match(a,b) (strcmp (a, b) == 0)

typedef struct
{
    int      type;
    char *   name;
}
    ID; /* to be hooked onto field userptr */

void set_field_id (f, type, name) /* associate type and name with field f */
FIELD * f;
int type;
char * name;
{
    /* allocate space, see malloc(3curses) */
    ID * id = (ID *) malloc (sizeof (ID));

    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 */
}
```

**Screen 11-6. Using the Field User Pointer to Match Items**

Note that if a match is not found, `find_field` returns a NULL field pointer. See the previous sections on panel and menu user pointers for more examples.

If successful, `set_field_userptr` returns `E_OK`. If not, it returns the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |

To change the system default user pointer from NULL to one of your choice, you need only pass `set_field_userptr` a NULL field pointer. Passing a NULL field pointer to `field_userptr` returns the current default user pointer.

## Manipulating Field Options

ETI provides several field options for controlling how data is entered and displayed in a field. The following functions let you set or clear these options or read their settings.

**SYNOPSIS**

```
int set_field_opts (field, opts)
FIELD * field;
OPTIONS opts;

OPTIONS field_opts (field)
FIELD * field;

options:
    O_VISIBLE
    O_ACTIVE
    O_PUBLIC
    O_EDIT
    O_WRAP
    O_BLANK
    O_AUTOSKIP
    O_NULLOK
    O_PASSOK
    O_STATIC
```

Function `set_field_opts` turns off all options that do not appear in its second argument. By default, all options are on.

The field options and their effects are as follows:

| | |
|---|---|
| `O_VISIBLE` | determines field visibility. If this option is on, the field is displayed. If this option is off, it is erased. This option is useful for supporting pop-up fields, fields visible or not depending on another field's value. |
| `O_ACTIVE` | determines if a field is visited during form processing. If inactivated, the field is ignored during form processing. Inactive fields enable you to create field labels and other static form symbols or changeable symbols that are not affected during form processing. Examples of fields that change value but are not affected during form processing are row and column totals, as in a spreadsheet program. You can change field values using calls to `set_field_buffer`. |
| `O_PUBLIC` | determines how feedback is presented to the user as data is entered. The data in public fields is displayed as entered, while the data in non-public fields is not displayed at all. Further, in non-public fields, the cursor does not actually move across the field, but the forms subsystem internally maintains the cursor position relative to the field data. You can use non-public fields to implement password fields. |

| | |
|---|---|
| `O_EDIT` | determines if field editing is permitted. By default, this option is on and a field may be edited. If the `O_EDIT` option is off, the field may be visited but not changed. Editing requests or attempts to enter data will fail. (`REQ_PREV_CHOICE` and `REQ_NEXT_CHOICE` requests, however, are honored, if they are defined for the field's type.) This is useful for creating fields for browsing such as scrollable help messages. |
| `O_WRAP` | determines if word wrapping occurs at the end of each line of the field. If any character of the word does not fit on the line as it is entered, the entire word is automatically moved to the beginning of the next line, if there is one. If the `O_WRAP` option is off, the word is split between the two lines. |
| `O_BLANK` | determines if the whole field is automatically erased when the end-user types a character in the first character position of the field before any character position has been changed. If the `O_BLANK` option is off, this does not occur. |
| `O_AUTOSKIP` | determines how the field responds when it becomes full. Ordinarily, when a field is full, an automatic request to move to the next field on the form is generated. If, however, the `O_AUTOSKIP` option is off, the end-user remains at the end of the field.<br><br>The `O_AUTOSKIP` option will be ignored if the option `O_STATIC` is off and there is no maximum growth specified for the field. On a growable field with no maximum growth specified, there is no “last” character position. If a maximum growth is specified, the `O_AUTOSKIP` option will cause an `REQ_NEXT_FIELD` to be generated from the last character position if the field has grown to its maximum size. |
| `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. |
| `O_STATIC` | When this option is on, the field is fixed in size and any attempt to add more data than the current field buffer will hold will fail. If it is off, the field will grow dynamically to accommodate additional data entered by the user. See “Dynamically Growable Fields” on page 11-9 for more information on dynamic fields. |

Remember that options are Boolean values. So to turn off option `O_ACTIVE` for field `f0` and to turn it on for field `f1`, you use the Boolean operators and write:

```
    FIELD * f0, * f1;
```

```
set_field_opts (f0, field_opts (f0) & ~O_ACTIVE);
            /* turn option off */
set_field_opts (f1, field_opts (f1) |  O_ACTIVE);
            /* turn option on  */
```

**NOTE**

Although you can change field option settings on posted forms, you cannot change option settings for the current field.

ETI also provides the following two functions which let you turn a field option on or off without using function `field_opts`.

**SYNOPSIS**

```
int field_opts_on (field, opts)
FIELD * field;
OPTIONS opts;

int field_opts_off (field, opts)
FIELD * field;
OPTIONS opts;
```

Unlike function `set_field_opts`, these functions leave unnamed option 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.

**SYNOPSIS**

```
FORM * new_form (fields)
FIELD ** fields;

int free_form (form)
FORM * form;
```

The function `new_form` takes as an argument a NULL-terminated, ordered array of FIELD pointers that define the fields on the form. The order of the field pointers determines the order in which the fields are visited during form driver processing discussed below.

As with the comparable ETI menu function `new_menu`, function `new_form` does not copy the array of field pointers. Instead, it saves the pointer to the array. Be sure not to change the array of field pointers once it has been passed to `new_form`, until the form is freed by `free_form` or the field array replaced by `set_form_fields` described in the next section.

Fields passed to `new_form` are connected to the resulting form.

**NOTE**

Fields may be connected to only one form at a time.

To connect fields to another form, you must first disconnect them using `free_form` or `set_form_fields`. If `fields` is NULL, the form is created but no fields are connected to it.

Unlike menus, ETI forms are logically divided into pages. Two functions enable you to mark a field that is to start a new page and to return a Boolean value indicating whether a given field does so.

**SYNOPSIS**

```
int set_new_page(field, bool)
FIELD * field;
int bool;            /* TRUE or FALSE */

int new_page(field)
FIELD * field;
```

The initial system default value of `new_page` is FALSE. This means that, unless specified with `set_new_page`, each field is assumed to continue the current page.

**NOTE**

In general, you should make the size of each form page smaller than the form's window size.

If function `set_new_page` executes successfully, it returns `E_OK.` If not, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_CONNECTED` | field connected to form |

Screen 11-7 shows how to create a simple two-page form.

```
FIELD * f[7];
FORM * form;

/* create fields as described in “Creating and Freeing Fields” on page 11-6 */

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 */
```

**Screen 11-7.  Creating a Form**

If successful, `new_form` returns a pointer to the new form. If there is no memory available for the form or one of the given fields is connected to another form, `new_form` returns NULL. Undefined results occur if the array of field pointers is not NULL-terminated.

The function `free_form` disconnects all fields and frees any space allocated for the form. Its argument is a form pointer previously obtained from `new_form`. The fields themselves are not automatically freed.

**NOTE**

You should free the fields comprising a form using `free_field` only after you free their form using `free_form`.

If successful, `free_form` returns `E_OK.` If not, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL form pointer |
| `E_POSTED` | form is posted |

Posting forms is described below.

As with panel, item, menu, and field pointers, form pointers should not be used once they are freed. If they are, undefined results occur.

## Manipulating Form Attributes

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

### Changing and Fetching the Fields on an Existing Form

Once you create a form with one set of fields using `new_form`, you can change the fields connected to it.

#### SYNOPSIS

```
int set_form_fields (form, fields)
FORM * form;
FIELD ** fields;

FIELD ** form_fields (form)
FORM * form;
```

Like `new_form`, function `set_form_fields` takes as an argument a NULL-terminated, ordered array of FIELD pointers that define the fields on the form and determine the order in which the fields are visited during form driver processing.

When `set_form_fields` is called, the fields previously connected to the form are disconnected from it (but not freed) before the new fields are connected. Like any set of fields connected to a form, the new fields cannot be passed to other forms while they are connected to the given form. You must first disconnect them by calling `free_form` or again calling `set_form_fields`.

There are two ways to disconnect the fields associated with a form without connecting another set of fields to the form:

- you can call `free_form`
- you can call `set_form_fields` with `fields` set to NULL

The first method frees the space allocated for the form, whereas the second does not.

To change the fields associated with `form` to those referenced in array pointer `new-fields`, 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 “Posting and Unposting Forms” on page 11-38.

The function `form_fields` returns the array of field pointers defining the form's fields. The function returns NULL if no fields are connected to the form or the form pointer is NULL.

## Counting the Number of Fields

The following function returns the number of fields connected to the given form.

**SYNOPSIS**

```
int field_count (form)
FORM * form;
```

If *form* is NULL, `field_count` returns -1.

As an example, consider the following routine, which determines whether your user is on the last field of the form as numbered in the field pointer array:

```
int on_last_field (form)
FORM * form;
{
    /* fetch number of last field */

    int lastindex = field_count (form) - 1;

    /* determine whether number of current field
        is the same */
        return field_index (current_field (form)) ==
            lastindex;
    }
```

Note the use of functions `field_index` and `current_field`, described in “Manipulating the Current Field” on page 11-57.

## Querying the Presence of Offscreen Data

It may be desirable to indicate to the user whether there is additional data either ahead or behind in a scrollable field. It is the responsibility of application developers to indicate, however they like, the presence of off screen data. The following functions allow the developer to query the presence of offscreen data.

**SYNOPSIS**

```
    int data_ahead(form)
    FORM *form;

    int data_behind(form)
    FORM *form;
```

`data_ahead` returns TRUE, if there is either more data offscreen to the right if the current field is a one line field, or more data offscreen below if the current field is multi-line. Otherwise FALSE is returned. Data is defined to be any non-pad character; see “Setting the Field Foreground, Background, and Pad Character” on page 11-19 for more detail on the pad character.

`data_behind` returns TRUE, if the first character position of the current field is not currently being displayed. Otherwise FALSE is returned.

## Changing ETI Form Default Attributes

During form initialization using `new_form`, all form attributes are assigned default values. As you can with menu attributes, you can change these default attribute values by calling the appropriate function with a NULL form pointer as its first argument. All subsequent forms created using `new_form` will then have the new default attribute value. However, forms created before the change to the current default value will retain the initial values of their attributes. Several examples of changing default values occur throughout the rest of this chapter.

## Displaying Forms

In general, to display a form, you determine the form dimensions, optionally associate a window and subwindow with the form, post the form, and refresh the screen.

## Determining the Dimensions of Forms

Every form is associated with a window and subwindow.

**NOTE**

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

Windows are used to create borders, titles, and the like. Before ETI posts a form, it must determine the sizes of its window and subwindow.

To determine the minimum window or subwindow size for a form, ETI considers the following:

- the number of rows and columns for each field
- the starting position (upper left corner) of each field within the form subwindow

By automatically fetching this information previously established by calls to `new_field`, function `scale_form` saves you the effort of calculating the size of your form subwindow.

### Scaling the Form

Considering the above information, this function returns the minimum window size necessary for containing the form.

#### SYNOPSIS

```
int scale_form (form, rows, cols)
FORM * form;
int * rows;
int * cols;
```

Because function `scale_form` must return more than one value (namely, the minimum number of rows and columns for the form) and C passes parameters “by value” only, the arguments of `scale_form` are pointers. The pointer arguments *rows* and *cols* point to locations used to return the minimum number of rows and columns for the form.

**NOTE**

You should call `scale_menu` only after the form's fields have been connected to the form using `new_form` or `set_form_fields`.

As an example, to return the minimum (sub)window size for form `f` in variables *rows* and *cols*, you can write:

```
FORM * form;
int rows, cols;

    /* create fields
        create form */

    /* determine minimum row and column size */
    scale_form (form, &rows, &cols);

   /* create form subwindow, as described
      in next section */
```

If function `scale_form` encounters an error, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL form pointer |
| `E_NOT_CONNECTED` | no fields connected to the form |

## Associating Windows and Subwindows with a Form

Remember that two windows are associated with every form — the form window and the form subwindow. The following functions assign windows and subwindows to forms and fetch those previously assigned to them.

### SYNOPSIS

```
int set_form_win (form, window)
FORM * form;
WINDOW * window;

WINDOW * form_win (form)
FORM * form;

int set_form_sub (form, window)
FORM * form;
WINDOW * window;

WINDOW * form_sub (form)
FORM * form;
```

These functions enable you to place stylistic borders, titles, and other decoration around a form.

**NOTE**

Remember that if the form window is NULL (the default), ETI uses `stdscr`. If the form subwindow is NULL (the default), ETI uses the form window so you need not use functions `set_form_win` or `set_form_sub` at all.

If you do not want to use `stdscr`, you should create a window and a subwindow for every form. ETI automatically writes all low-level ETI (**`curses`**) output of the form proper on the form subwindow. If you want further output (such as borders, titles, or static messages), you should write it on the form window. However, you need not write any further output at all.

**NOTE**

Be sure to apply all low—level ETI (**`curses(3curses)`**) command output and refresh operations to your form's window, not its subwindow.

Figure 11-1 diagrams the relationship between ETI Form functions, your application program, and its form window and subwindow.

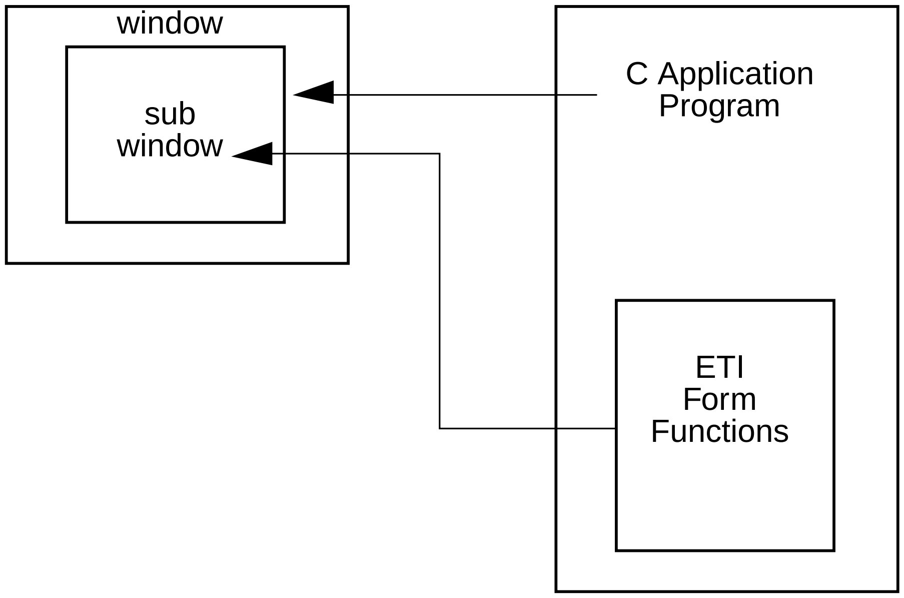


**Figure 11-1.  Form Functions Write to Subwindow, Application to Window**

Screen 11-8 shows how to create a form with a border of the terminal's default vertical and horizontal characters.

```
    /*  create window 4 characters larger than form dimensions
    with top left corner at (0, 0).  subwindow is positioned
    at (2, 2) relative to the form window origin with dimensions
    equal to the form dimensions.  */

FORM * f;
WINDOW * w;
int rows, cols;

scale_form (f, &rows, &cols); /* get dimensions of form */

if (w = newwin (rows+4, cols+4, 0, 0))
{
    set_form_win (f, w);  /* associate window and subwindow with form */

    set_form_sub (f, derwin (w, rows, cols, 2, 2));

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

**Screen 11-8.  Creating a Border around a Form**

Function `scale_form` sets the values of the variables `rows` and `cols`, which provide the form dimensions without the border. Adding four to the dimensions of the form window clearly sets off the form border from the fields of the form (the form proper).

If functions `set_form_win` or `set_form_sub` encounter an error, they return one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_POSTED` | form is posted |

As usual, you can change the default form window or subwindow. For instance, you can change the default form window from `stdscr` to a window `w` by passing a NULL form pointer, as follows:

```
int rows, cols, firstrow, firstcol;

    /* create form window */

WINDOW * w = newwin (rows, cols, firstrow, firstcol);

set_form_win((FORM *)0, w);
    /* change default form window to w */
```

Note that if you later change a posted form by writing directly to its window, before continuing you must reposition the form window cursor using `pos_form_cursor`. See “Positioning the Form Cursor” on page 11-60.

## Posting and Unposting Forms

When you have created a form and its window and subwindow, you are ready to post it. To post a form is to display it on the form's subwindow; to unpost a form is to erase it from the form's subwindow.

**SYNOPSIS**

```
int post_form (form)
FORM * form;

int unpost_form (form)
FORM * form;
```

Unposting a form does not remove its data structure from memory.

**NOTE**

To post a form, be sure that you have connected fields to it first.

Screen 11-9 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 Screen 11-8 to create the form's window and subwindow.

```
static void display_form (f)/* create form windows and post */
FORM * f;
{
    WINDOW *w;
    int     rows;
    int     cols;

    scale_form (f, &rows, &cols);/* get dimensions of form */

    /*  create form window as in Screen 11-8  */

    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 to
           High-level Functions” */
        error (“error return from newwin”, NULL);

    if (post_form (f) != E_OK)      /* post form */

        error (“error return from post_form”, NULL);
    else
        refresh (w);
}

static void erase_form (f)/* unpost and delete form windows */
FORM * f;
{
    WINDOW * w = form_win (f);
    WINDOW * s = form_sub (f);

    unpost_form (f);       /* unpost form */
    werase (w);           /* erase form window */
    wrefresh (w);         /* refresh screen */
    delwin (s);           /* delete form windows */
    delwin (w);
}
```

**Screen 11-9. Posting and Unposting a Form**

If successful, function `post_form` returns `E_OK.` If not, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL form pointer |
| `E_POSTED` | form is already posted |
| `E_NOT_CONNECTED` | no connected fields |
| `E_NO_ROOM` | form does not fit in subwindow |

If successful, the function `unpost_form` returns `E_OK.` If not, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL form pointer |

| | |
|---|---|
| `E_NOT_POSTED` | form is not posted |
| `E_BAD_STATE` | called from init/term function |

The initialization and termination routines are discussed in the next section.

## Form Driver Processing

Like the function `menu_driver` for the menu subsystem, function `form_driver` is the workhorse of the form system. Once the form is posted, the form driver handles all interaction with your end-user. The form driver responds to

- field navigation requests
- page navigation requests
- field editing requests
- data entry
- field validation requests

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

**SYNOPSIS**

```
int form_driver (form, c)
FORM * form;
int c;
```

As with menu processing, to enable the form driver to process your end-users' requests, you must write an input key virtualization routine. This 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 Screen 11-10, 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 “Calling the Form Driver” on page 11-48.

```
    /*  The following key mapping is defined by get_request.
    Note that ^X represents the character control-X.

        ^Q      - end form processing

        ^F      - move to next page
        ^B      - move to previous page
        ^N      - move to next field
        ^P      - move to previous field
        home key- move to first field
        home down- move to last field
        ^L      - move left to field
        ^R      - move right to field
        ^U      - move up to field
        ^D      - move down to field

        ^W      - move to next word
        ^T      - move to previous word
        ^S      - move to beginning of field data
        ^E      - move to end of field data
        left arrow- move left in field
        right arrow- move right in field
        down arrow- move down in field
        up arrow- move up in field

        ^M <CR>  - enter new line
        ^I      - insert blank character
        ^O      - insert blank line
        ^V      - delete character
        ^H <BS>  - delete previous character
        ^Y      - delete line
        ^G      - delete word
        ^C      - clear to end of line
        ^K      - clear to end of field
        ^X      - clear entire field
        ^A      - request next field choice
        ^Z      - request previous field choice
        ESC     - toggle between insert and overlay mode

    define application commands   */


#define QUIT(MAX_COMMAND + 1)

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

    switch (c)
    {
```

**Screen 11-10.  A Sample Key Virtualization Routine**

```
        case 0x11:   /* ^Q */ returnQUIT;

        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 “Field Editing Requests” on page 11-45.

Next, `get_request` calls `wgetch` to read a character entered by the user. The `switch` statement maps the character read onto a specific application command or form request. The application command QUIT appears here as the first case; the other cases map characters onto form requests. Any character that is not an application command or form request is simply returned unchanged—it is treated as data being entered into the current field.

Note that this key mapping assumes your end-user will be using a terminal with arrow keys (KEY_LEFT, KEY_RIGHT, KEY_UP, KEY_DOWN), a home key (KEY_HOME), and a home down key (KEY_LL).

## ETI Form Requests

The ETI form subsystem places the following requests at your application program's disposal.

### Page Navigation Requests

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

| | |
|---|---|
| `REQ_NEXT_PAGE` | move to next page |
| `REQ_PREV_PAGE` | move to previous page |
| `REQ_FIRST_PAGE` | move to first page |
| `REQ_LAST_PAGE` | move to last page |

Page navigation requests are cyclic so that

- the `REQ_NEXT_PAGE` request from the last page moves to the first page
- the `REQ_PREV_PAGE` from the first page moves to the last.

### Inter-field Navigation Requests on the Current Page

These requests enable your end-user to move from field to field on the current page of a single form.

| | |
|---|---|
| `REQ_NEXT_FIELD` | move to next field |
| `REQ_PREV_FIELD` | move to previous field |
| `REQ_FIRST_FIELD` | move to first field |
| `REQ_LAST_FIELD` | move to last field |
| `REQ_SNEXT_FIELD` | move to sorted next field |
| `REQ_SPREV_FIELD` | move to sorted previous field |
| `REQ_SFIRST_FIELD` | move to sorted first field |
| `REQ_SLAST_FIELD` | move to sorted last field |
| `REQ_LEFT_FIELD` | move left to field |
| `REQ_RIGHT_FIELD` | move right to field |
| `REQ_UP_FIELD` | move up to field |
| `REQ_DOWN_FIELD` | move down to field |

All field navigation requests are cyclic on the current page so that

- the `REQ_NEXT_FIELD` request from the last field on a page moves to the first field on that page.
- the `REQ_PREV_FIELD` request from the first field on a page moves to the last field on that page.

and so forth. The order of the fields in the field array passed to `new_form` determines the order in which the fields are visited using the `REQ_NEXT_FIELD`, `REQ_PREV_FIELD`, `REQ_FIRST_FIELD`, and `REQ_LAST_FIELD` requests.

**NOTE**

Remember that the order of fields in the form array is simply the order in which fields are processed during form processing. This order bears no necessary relation to the order of the fields as they are displayed on the form page.

Your end-user may also move from field to field on the form page in row-major order — left to right, top to bottom. To do so, you use the `REQ_SNEXT_FIELD`, `REQ_SPREV_FIELD`, `REQ_SFIRST_FIELD`, and `REQ_SLAST_FIELD` requests.

Finally, your end-user can move about in different directions using the `REQ_LEFT_FIELD`, `REQ_RIGHT_FIELD`, `REQ_UP_FIELD`, and `REQ_DOWN_FIELD` requests. Note that the first character (top left corner) of the field is used to determine where the field is located relative to other fields. This means, for example, that a multi-line field whose first character is on the second row of a form is not on the same row as a field whose first character is on the third row of a form even though the multi-line field may extend below the third row.

## Intra-field Navigation Requests

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

| | |
|---|---|
| `REQ_NEXT_CHAR` | move to next character in field |
| `REQ_PREV_CHAR` | move to previous character in field |
| `REQ_NEXT_LINE` | move to next line in field |
| `REQ_PREV_LINE` | move to previous line in field |
| `REQ_NEXT_WORD` | move to next word in field |
| `REQ_PREV_WORD` | move to previous word in field |
| `REQ_BEG_FIELD` | move to beginning of field |
| `REQ_END_FIELD` | move after last character in field |
| `REQ_BEG_LINE` | move to beginning of line |
| `REQ_END_LINE` | move after last character in line |

| | |
|---|---|
| `REQ_LEFT_CHAR` | move left in field |
| `REQ_RIGHT_CHAR` | move right in field |
| `REQ_UP_CHAR` | move up in field |
| `REQ_DOWN_CHAR` | move down in field |

The effect of these requests is as follows:

- The `REQ_NEXT_CHAR` and `REQ_PREV_CHAR` requests step forward and backward through the field.
- The `REQ_NEXT_LINE` and `REQ_PREV_LINE` requests move the cursor to the beginning of the next and previous line.
- The `REQ_NEXT_WORD` and `REQ_PREV_WORD` requests move the cursor to the beginning of the next or previous word.
- The `REQ_BEG_FIELD` places the cursor at the first non-pad character in the field. The `REQ_END_FIELD` request places the cursor after the last non-pad character in the field. This lets the user easily add characters to the field. If there is no room, it returns the cursor to the start of the field.
- The `REQ_BEG_LINE` request places the cursor at the first non-pad character in the current line of the field. The `REQ_END_LINE` request places the cursor after the last non-pad character in the current line. If there is no room, it returns the cursor to the start of the line.
- The `REQ_LEFT_CHAR`, `REQ_RIGHT_CHAR`, `REQ_UP_CHAR`, and `REQ_DOWN_CHAR` requests move one character position in the stated direction.

## Field Editing Requests

These requests set the editing mode — insert or overlay.

| | |
|---|---|
| `REQ_INS_MODE` | begin insert mode |
| `REQ_OVL_MODE` | begin overlay mode |

In insert mode (the default), all text is inserted at the current cursor position, while all existing text starting at the current cursor position is moved to the right. In overlay mode, text entered by your end-user overlays (replaces) existing text in the field. In both modes, the cursor is advanced one character position as each character is entered.

The following requests provide a complete set of field editing requests.

| | |
|---|---|
| `REQ_NEW_LINE` | new line request |
| `REQ_INS_CHAR` | insert blank character at cursor |
| `REQ_INS_LINE` | insert blank line at cursor |
| `REQ_DEL_CHAR` | delete character at cursor |
| `REQ_DEL_PREV` | delete character before cursor |

| | |
|---|---|
| `REQ_DEL_LINE` | delete line at cursor |
| `REQ_DEL_WORD` | delete word at cursor |
| `REQ_CLR_EOL` | clear to end of line |
| `REQ_CLR_EOF` | clear to end of field |
| `REQ_CLR_FIELD` | clear entire field |

The effects of `REQ_NEW_LINE` and `REQ_DEL_PREV` requests depend on several factors such as the current mode (insert or overlay) and the cursor position within the field.

- The effects of `REQ_NEW_LINE` are as follows:
  - In insert mode — if the cursor is at the beginning of a field or on the last line of a field, the `REQ_NEW_LINE` request acts like a `REQ_NEXT_FIELD` request. Otherwise, the `REQ_NEW_LINE` request inserts a new line after the current line and moves the text on the current line starting at the cursor position to the beginning of the new line. The cursor is moved to the beginning of the new line.
  - In overlay mode — if the cursor is at the beginning of a field, the `REQ_NEW_LINE` request acts like a `REQ_NEXT_FIELD` request. If the cursor is on the last line of a field, the `REQ_NEW_LINE` request erases all data from the cursor position to the end of the line and satisfies a `REQ_NEXT_FIELD` request. Otherwise, the `REQ_NEW_LINE` request erases all data from the cursor position to the end of the line and moves the cursor to the beginning of the next line.
- The effects of the `REQ_DEL_PREV` request is as follows:
  - In insert mode — if the cursor is at the beginning of a field, the `REQ_DEL_PREV` request behaves like a `REQ_PREV_FIELD` request. If the cursor is at the beginning of a line other than the first and the text on that line will fit at the end of the preceding line, the text is moved and the current line is deleted. Otherwise, the `REQ_DEL_PREV` request simply deletes the previous character.
  - In overlay mode — if the cursor is positioned at the beginning of a field, the `REQ_DEL_PREV` request behaves like a `REQ_PREV_FIELD` request. Otherwise, the `REQ_DEL_PREV` request simply deletes the previous character.

Because the requests `REQ_NEW_LINE` and `REQ_DEL_PREV` automatically do a request `REQ_NEXT_FIELD` or `REQ_PREV_FIELD` as described, they are said to be overloaded field editing requests. See the remarks on options `O_NL_OVERLOAD` and `O_BS_OVERLOAD` in “Setting and Fetching Form Options” on page 11-62.

## Scrolling Requests

Fields can scroll if they have offscreen data. A field can have offscreen data if it was originally created with offscreen rows-the parameter *nrow* in the `new_field` library routine was greater than 0-or the field has grown larger than its original size. See “Dynamically Growable Fields” on page 11-9 for more details on the growth of fields.

There are two kinds of scrolling fields, vertically scrolling fields and horizontally scrolling fields. Multi-line fields with offscreen data scroll vertically and one line fields with off-screen data scroll horizontally. Recall the library routine `new_field`; a new field created with *rows* set to one and *nrow* set to zero will be defined to be a one line field. A new field created with *rows* `+` *nrow* greater than one will be defined to be a multi-line field.

The following form driver requests are used on vertically scrolling multi-line fields.

| | |
|---|---|
| `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 |
| `REQ_SCR_FHPAGE` | scroll field forward half a page |
| `REQ_SCR_BHPAGE` | scroll field backward half a page |

In the descriptions above, a page is defined to be the number of visible rows of the field as displayed on the form.

The following form driver requests are used on horizontally scrolling one line fields.

| | |
|---|---|
| `REQ_SCR_FCHAR` | scroll field forward a character |
| `REQ_SCR_BCHAR` | scroll field backward a character |
| `REQ_SCR_HFLINE` | scroll field forward a line |
| `REQ_SCR_HBLINE` | scroll field backward a line |
| `REQ_SCR_HFHALF` | scroll field forward half a line |
| `REQ_SCR_HBHALF` | scroll field backward half a line |

In the descriptions above, a line is defined to be the width of the field as displayed on the form.

In addition, intra-field navigation requests may generate implicit scrolling on scrollable fields. See “Intra-field Navigation Requests” on page 11-44.

## 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 “Setting and Reading Field Buffers” on page 11-21 and “Setting and Reading the Field Status” on page 11-22.) 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 “Setting the Field Type to Ensure Validation” on page 11-13 and “Creating and Manipulating Programmer-defined Field Types” on page 11-65 for information on these field types.

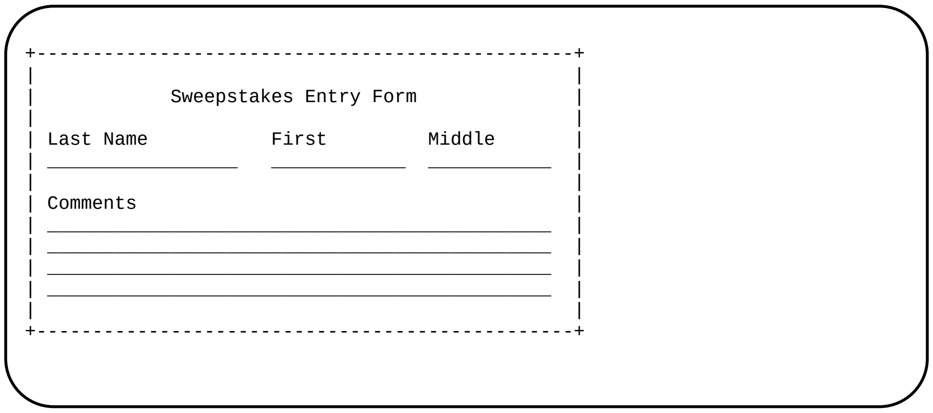
## Application-defined Commands

Form requests are implemented as integers above the low-level ETI (`curses`) maximum key value `KEY_MAX`. A symbolic constant `MAX_COMMAND` is provided so applications can implement their own commands without conflicting with the ETI form or menu subsystems. All ETI system form requests are greater than `KEY_MAX` and less than or equal to `MAX_COMMAND`. You should set your application-defined commands to an integer greater than `MAX_COMMAND`.

## Calling the Form Driver

The ETI form driver works very much like the ETI menu driver. As soon as the form driver receives a request, it checks if it is an ETI form request. If so, it performs the request and reports the results. If the request is not an ETI form request, the form driver checks if the character is data, 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 Screen 11-2.



```
+------------------------------------------------+
|                                                |
|            Sweepstakes Entry Form              |
|                                                |
| Last Name          First         Middle        |
| _________________  ____________  ___________   |
|                                                |
| Comments                                       |
| ____________________________________________   |
| ____________________________________________   |
| ____________________________________________   |
| ____________________________________________   |
|                                                |
+------------------------------------------------+
```

**Figure 11-2. Sweepstakes Form Output**

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

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

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

static void start_curses()/* see the previous section “ETI Low-level */
                 /* Interface to High-level Functions” */

static void display_form (f)/* create form windows and post */
                 /* see Screen 11-9 for details */


static void erase_form (f)/* unpost and delete form windows */
                 /* see Screen 11-9 for details */

    /*  define application commands  */

#define QUIT (MAX_COMMAND + 1)

static int get_request (w)/* virtual key mapping; see Screen 11-10 */


static int my_driver (form, c)/* handle application commands */
FORM * form;
int c;
{
    switch (c)
    {
        case QUIT:

    /* validate current field */

            if (form_driver (form, REQ_VALIDATION) == E_OK)
                return TRUE;
            break;
    }
    beep ();/* signal error */
    return FALSE;
}

main (argc, argv)
int argc;
char * argv[];
{
    WINDOW *w;
    FORM *   form;
    FIELD **f;
    FIELD **make_fields ();
    void     free_fields ();
    int      c, done = FALSE;

    PGM = argv[0];
```

**Screen 11-11.  An Example of Form Driver Usage**

```
    if (! (form = new_form (make_fields ())))
        error (“error return from new_form”, NULL);

    start_curses ();
    display_form (form);

    /*  interact with user  */

    w = form_win (form);

    while (! done)
    {
        switch (form_driver (form, c = get_request (w)))
        {
            case E_OK:
                break;
            case E_UNKNOWN_COMMAND:
                done = my_driver (form, c);
                break;
            default:
                beep ();/* signal error */
                break;
        }
    }

    /* terminate form processing */

    erase_form (form);
    end_curses ();
    f = form_fields (form);
    free_form (form);
    free_fields (f);
    exit (0);
}

typedef FIELD *(* PF_field ) ();

typedef struct           /* define struct for creation */
{
    PF_fieldtype;/* 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_FIELD512

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++);
}
```

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 what we have seen previously. (Array `F` holds the string labels and field sizes; it can be changed so that `make_fields` can create any form.) Function `main` then initializes **`curses`** using `start_curses` and displays the form using `display_form`.

In its `while` loop, `main` repeatedly calls `form_driver` with the character returned by `get_request`. If the form driver does not recognize the character as a request or data, it returns `E_UNKNOWN_COMMAND,` whereupon the application-defined routine `my_driver` is called with the same character. Routine `my_driver` processes the application-defined commands. In this example, there is only one, QUIT. Note how this request automatically calls the form driver again, now with the `REQ_VALIDATION` request. Remember that this request is necessary to ensure that current field validation occurs before your end-user leaves the form. If validation is successful, `my_driver` returns TRUE. In turn, this sets `done` to TRUE, and the while loop is exited.

Finally, `main` erases the form, terminates low-level ETI (**`curses`**), frees the form and its fields, and exits the program.

This example is typical, but it is only one of many ways you can structure an application. ETI's flexibility lets you use it over a wide range of applications.

Like other ETI routines that return an `int`, the form driver returns `E_OK` if it recognizes and processes the input character argument. If it encounters an error, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL form pointer |
| `E_BAD_STATE` | called from init/term routines |
| `E_NOT_POSTED` | form is not posted |
| `E_UNKNOWN_COMMAND` | unknown command |
| `E_REQUEST_DENIED` | recognized request failed |
| `E_INVALID_FIELD` | failed field validation |

**NOTE**

Like the menu driver, the form driver may not be called from any of the initialization or termination routines described next. Any attempt to do so returns `E_BAD_STATE.`

## Establishing Field and Form Initialization and Termination Routines

As with the menu driver, you may sometimes want the form driver to execute a specific routine whenever the current field or form changes. The following routines let you do this.

### SYNOPSIS

```
typedef void (* PTF_void ) ();

int set_form_init (form, func)
FORM * form;
PTF_void func;
PTF_void form_init (form)
FORM * form;

int set_form_term (form, func)
FORM * form;
PTF_void func;

PTF_void form_term (form)
FORM * form;

int set_field_init (form, func)
FORM * form;
PTF_void func;

PTF_void field_init (form)
FORM * form;

int set_field_term (form, func)
FORM * form;
PTF_void func;

PTF_void field_term (form)
FORM * form;
```

The argument *func* is a pointer to the specific function you want executed by the form driver. This application-defined function itself takes a form pointer as an argument.

As with menus, if you want your application to execute a routine at one of the initialization or termination points listed below, you should call the appropriate form initialization or termination routine at the start of your program. If you do not want a specific function called in these cases, you may refrain from calling these routines altogether.

## Function set_form_init

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

- when the form is posted
- just after every form page operation, that is, after the page changes on a posted form

## Function set_field_init

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

- when the form is posted
- just after a field change operation, that is, every time the current field changes on a posted form.

## Function set_field_term

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

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

## Function set_form_term

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

- just before every form page operation, 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, Screen 11-12 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.

```
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 */
}
```

**Screen 11-12.  Sample Termination Routine that Updates a Column Total**

Function `set_field_buffer` sets the column total field to the value `total` stored in `buf`. See “Setting and Reading Field Buffers” on page 11-21 for details on `field_buffer` and `set_field_buffer`.

For another example, consider Screen 11-13. 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.

```
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);
}
```

**Screen 11-13. Field Initialization and Termination to Highlight Current Field**

If functions `set_form_init`, `set_form_term`, `set_field_init`, or `set_field_term` encounter an error, they return the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |

As usual, if you want a specific default initialization or termination function for all forms or all fields, you can pass the appropriate set function a NULL form pointer. Passing a NULL form pointer to the access functions returns the current ETI default.

## Manipulating the Current Field

The current field is the field where your end-user is positioned on the display screen. It changes as the end-user moves about the form entering or changing data. The cursor rests on the current field. To have your application program set or determine the current field, you use the following functions.

**SYNOPSIS**

```
int set_current_field (form, field)
FORM * form;
FIELD * field;

FIELD * current_field (form)
FORM * form;
```

```
int field_index (field)
FIELD * field;
```

The function `set_current_field` enables you to set the current field, while function `current_field` returns the pointer to it. The value returned by `field_index` is the index to the given field in the field pointer array associated with the connected form. This value is in the range of 0 through `N-1`, where `N` is the total number of fields.

When a form is created by `new_form` or the fields associated with the form are changed by `set_form_fields` the current field is automatically set to the first visible, active field on page 0.

**NOTE**

Your application program need not call `set_current_field` unless you want to implement field navigation requests that are not supported by the form driver and discussed in “ETI Form Requests” on page 11-43.

Screen 11-14 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.

```
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;
}
```

**Screen 11-14.  Example Manipulating the Current Field**

If function `set_current_field` encounters an error, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL form pointer or field not connected to form |
| `E_BAD_STATE` | called from init/term routines |
| `E_INVALID_FIELD` | current field is invalid on posted form |
| `E_REQUEST_DENIED` | field not active or not visible |

The function `current_field` returns (FIELD *) 0 if given a NULL form pointer or there are no fields connected to the form.

The function `field_index` returns -1 if its field pointer argument is NULL or the field is not connected to a form.

## Changing the Form Page

Two form functions enable your application program to change to another page on the form or to determine the current page of the form.

**SYNOPSIS**

```
int set_form_page (form, page)
FORM * form;
int page;

int form_page (form)
FORM * form;
```

Upon execution of `set_form_page`, the current field is set to the first field on the new page that is visible and active (visited during form driver processing). Variable *page* must be in the range of 0 through `N-1`, where `N` is the total number of pages. The function `form_page` returns the page number of the page currently visible on the screen.

When function `new_form` creates a form or function `set_form_fields` changes the fields associated with a form, the form page is automatically set to 0.

**NOTE**

Your application program need not call `set_form_page` unless you want to implement page navigation requests that are not supported by the form driver and discussed in “ETI Form Requests” on page 11-43.

Screen 11-15 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.

```
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 */
}
```

**Screen 11-15.  Example Changing and Checking the Form Page Number**

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

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL form or page out of range |
| `E_BAD_STATE` | called from init/term routines |
| `E_INVALID_FIELD` | current field is invalid on posted form |

The function `form_page` returns -1 if given a NULL form pointer or there are no fields connected to the form.

## Positioning the Form Cursor

As with menu processing, some processing of user form requests may move the cursor from the location required for continued processing by the form driver. This function moves the cursor back to where it belongs.

### SYNOPSIS

```
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.

Screen 11-16 illustrates one use of this function. Function `printpage` repositions the cursor after it prints the page number in the form window.

```
void printpage (form)
FORM * form;
{
    int     p = form_page (form) + 1;
    WINDOW *w = form_win (form);
    int     rows, cols;
    char    buf[80];

    box (w, 0, 0);              /* put border around form window */
    getmaxyx (w, rows, cols);   /* fetch window size */
    sprintf (buf, " %d ", p);   /* store next page number */


    wmove (w, (rows-1), ((cols-1)-strlen(buf))/2); /* position cursor */
    waddstr (w, buf);         /* print page number */

    /* position the form cursor for continued form processing */

    pos_form_cursor (form);
}

main ()
{
    FORM * form;

    set_form_init (form, printpage);
}
```

**Screen 11-16. Repositioning the Cursor after Printing Page Number**

If `pos_form_cursor` encounters an error, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL form pointer |
| `E_NOT_POSTED` | form is not posted |

## Setting and Fetching the Form User Pointer

As it does for items, menus, and fields, ETI supplies a form user pointer for data such as titles, help messages, and the like. These functions enable you to set the pointer and return its referent.

**SYNOPSIS**

```
int set_form_userptr (form, userptr)
FORM * form;
char * userptr;

char * form_userptr (form)
FORM * form;
```

You can define a structure to be connected to the form using this pointer. By default, the form user pointer is NULL.

Screen 11-17 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.

```
#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 */
}
```

**Screen 11-17.  Pattern Match Example Using Form User Pointer**

For more user pointer examples, see the previous sections on item, menu, and field user pointers and the sample programs at the end of this guide.

If successful, `set_form_userptr` returns `E_OK.` If not, it returns the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |

As usual, you change the default by passing `set_form_userptr` a NULL form pointer. So to change the default user pointer to point to the string `***`, you write:

```
/* change default user pointer */
set_form_userptr((form *) 0, “***”);
```

## Setting and Fetching Form Options

ETI provides form options regulating how specific user requests are handled. These functions enable you to set the options and read their settings.

**SYNOPSIS**

```
int set_form_opts (form, opts)
FORM * form;
```

```
    OPTIONS opts;

    OPTIONS form_opts (form)
    FORM * form;

    options:
        O_NL_OVERLOAD
        O_BS_OVERLOAD
```

Note that function `set_form_opts` automatically turns off all form options not referenced in its second argument. By default, all options are on.

The effects of the options are as follows:

| | |
|---|---|
| `O_NL_OVERLOAD` | determines how a `REQ_NEW_LINE` request is processed. If `O_NL_OVERLOAD` is on, the request is overloaded. See "Field Editing Requests" on page 11-45 for a description of overloading. If `O_NL_OVERLOAD` is off, the `REQ_NEW_LINE` request behavior depends on whether insert mode is on.<br><br>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.<br><br>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.<br><br>If the field option `O_STATIC` if off and there is no maximum growth specified for the field, the overloaded form driver request `REQ_NEW_LINE` will operate the same way regardless of the setting of the `O_NL_OVERLOAD` form option. If a field can grow without bound, there is no last line, so `REQ_NEW_LINE` will never implicitly generate a `REQ_NEXT_FIELD`. If a maximum growth limit is specified and the `O_NL_OVERLOAD` form option is on, `REQ_NEW_LINE` will only implicitly generate `REQ_NEXT_FIELD` if the field has grown to its maximum size and the user is on the last 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 "Field Editing Requests" on page 11-45 for information on overloading. If `O_BS_OVERLOAD` is off, the `REQ_DEL_PREV` request depends on whether insert mode is on.<br><br>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.

**SYNOPSIS**

```
int form_opts_on (form, opts)
FORM * form;
OPTIONS opts;

int form_opts_off (form, opts)
FORM * form;
OPTIONS opts;
```

Unlike function `set_form_opts`, these functions do not affect options unreferenced in their second argument.

Another way to turn off option `O_NL_OVERLOAD` on form `f0` and turn it on on form `f1` is to write

```
FORM * f0, * f1;

form_opts_off (f0, O_NL_OVERLOAD); /* turn option off */
form_opts_on  (f1, O_NL_OVERLOAD); /* turn option on */
```

If functions `set_form_opts`, `form_opts_off`, or `form_opts_on` encounter an error, they return the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |

To change the current system default from, say, `O_NL_OVERLOAD` to not-`O_NL_OVERLOAD` without affecting the `O_BS_OVERLOAD` option, you write:

```
form_opts_off( (FORM *) 0, O_NL_OVERLOAD);
```

## Creating and Manipulating Programmer-defined Field Types

In addition to the wealth of field types that ETI automatically provides, ETI lets you create new field types from old ones. For most applications, you may not need them, but when you do, you will have them.

### Building a Field Type from Two Other Field Types

One way to define a new field type is to create one from two existing field types. The function `link_fieldtype` lets you do this.

**SYNOPSIS**

```
FIELDTYPE * link_fieldtype(type1,type2)
FIELDTYPE * type1;
FIELDTYPE * type2;
```

The constituent types may be system-defined or programmer-defined types. They may require additional arguments for the later call to `set_field_type` and may be associated with validation functions or choice functions. Validation functions validate the value in the field, while choice functions enable the user to choose the next or previous value of the field type. See “Creating a Field Type with Validation Functions” on page 11-66 and “Supporting Next and Previous Choice Functions” on page 11-72.

If additional arguments are required for the later call to `set_field_type`, those of *type1* should precede those of *type2*. If there are validation or choice functions associated with the constituent types, the new type first executes the function associated with *type1*. If it is successful, it returns TRUE. If not, the new type executes the function associated with *type2*. Whatever it returns is the value returned by the new type.

As an example, the following code creates a new field type that accepts either a color keyword or an integer between 0 and 255, inclusive:

```
FIELD *f1;

extern char ** colors;

ENUM_OR_INT = link_fieldtype
    (TYPE_ENUM, TYPE_INTEGER);
       /* Constituent types are System types
          described in “Setting the Field Type
          to Ensure Validation” */

set_field_type (f1, ENUM_OR_INT, colors,
    FALSE, FALSE, 0, 0L, 255L);
       /* create field of field type
          ENUM_OR_INT */
```

Once you have created the new field type, you can create fields of that type. The last statement here creates field `f1`, which accepts only values of type `ENUM_OR_INT`.

If an error occurs, `link_fieldtype` returns the following:

| | |
|---|---|
| `NULL` | no available memory |

## Creating a Field Type with Validation Functions

Another way to create a new field type is by specifying

- a function that validates each character as it is entered into the field
- a function that validates the entire value entered into the field

or both. Function `new_fieldtype` returns your new field type given pointers to these validation functions.

### SYNOPSIS

```
typedef int (* PTF_int) ();

FIELDTYPE * new_fieldtype (f_check, c_check)
PTF_int f_check;
PTF_int c_check;
```

The form driver automatically calls the named validation functions during form driver processing.

To create a new field type, you must write at least one of the two 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_fieldtype` is useful for creating field types for specialized applications. For example, Screen 11-18 defines a new field type `TYPE_HEX` as a hex number between `0x0000` and `0xffff`.

```
#include <ctype.h>
#include <form.h>
extern long strtol ();

#define isblank(c) ((c) == ' ')

static int padding = 4;     /* pad on left to 4 digits  */
static long vmin = 0x0000L; /* minimum acceptable value */
static long vmax = 0xffffL; /* maximum acceptable value */

static int fcheck_hex (f, arg)
FIELD * f;
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 next section */
{
    return isxdigit (c);
}
FIELDTYPE * TYPE_HEX = new_fieldtype (fcheck_hex, ccheck_hex);
            /* create new field type */
```

**Screen 11-18. Creating a Programmer-defined Field Type**

Later, you assign fields with the field type `TYPE_HEX` as you do with any field type and field:

```
FIELD * field;

set_field_type (field, TYPE_HEX);
```

Function `ccheck_hex` checks that the input character is a valid 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.

**SYNOPSIS**

```
int free_fieldtype (fieldtype)
FIELDTYPE * fieldtype;
```

You may want to free the field type `TYPE_HEX` from the previous example once fields of that type have been processed. To do so, you write

```
/* create field type TYPE_HEX */
    create fields of this type
    free fields of this type */

free_fieldtype(TYPE_HEX);
    /* free programmer-defined type */
```

If successful, function `free_fieldtype` returns `E_OK.` If an error occurs, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | NULL field type |
| `E_CONNECTED` | type is connected to one or more fields |

Once a field type is freed, you must not use it again. If you do, the effect is undefined.

## Supporting Programmer-defined Field Types

You may want to support some programmer-defined field types with additional arguments or with previous and next choice functions. This section explains how to do so.

## Argument Support for Field Types

Some field types may require additional arguments to the `set_field_type` routine, which sets the field type of a field. Function `set_fieldtype_arg` takes as arguments pointers to functions that manage storage for the additional arguments.

SYNOPSIS

```
typedef char * (* PTF_charP) ();
typedef void   (* PTF_void) ();

int set_fieldtype_arg (fieldtype, make_arg, copy_arg, free_arg)
FIELDTYPE * fieldtype;
PTF_charP make_arg;
PTF_charP copy_arg;
PTF_void free_arg;
```

You must write the functions referenced by pointers *make_arg*, *copy_arg*, and *free_arg*. These functions should do the following:

| | |
|---|---|
| *make_arg* | allocate a structure for the field specific parameters to `set_field_type` and return a pointer to the saved data |
| *copy_arg* | duplicate the structure created by *make_arg* |
| *free_arg* | free any storage allocated by *make_arg* or *copy_arg* |

Function *make_arg* is called automatically when your application program calls `set_field_type`. It takes one argument, a `va_list *`. (See **`varargs(5)`** for details.) Function *make_arg* in turn should call `va_arg` for each additional argument to `set_field_type` associated with the field type. Note that function `va_start` is called by `set_field_type` before *make_arg* gains control, while function `va_end` is called by `set_field_type` after *make_arg* returns.

Function *make_arg* must allocate space for the information associated with the additional arguments, save the information, and return the pointer to the information cast to a character pointer. It is this character pointer that is the argument *arg* to the other functions associated with the field type, namely `copy_arg`, `free_arg`, `f_check`, `c_check`, `next_choice`, and `prev_choice`.

Function *copy_arg* takes as its sole argument a pointer to existing argument information. It returns a pointer to a copy of this information. Function *free_arg* takes as its sole argument a pointer to existing argument information. It should free any space allocated by *make_arg*.

Screen 11-19 illustrates how you can add padding and range arguments to our `TYPE_HEX` defined above.

```
/* TYPE_HEX
    set_field_type (f, TYPE_HEX, padding, vmin, vmax);

    int padding;      for padding with leading zeros
    long vmin;        minimum acceptable value
    long vmax;        maximum acceptable value */

#include <form.h>
#include <ctype.h>
#include <varargs.h>
extern long strtol ();

#define isblank(c) ((c) == ' ')

typedef struct {
    int padding;
    long vmin, vmax;
} HEX;

static char * make_hex (ap)
va_list * ap;
{
    HEX * n = (HEX *) malloc (sizeof (HEX));

    if (n)
    {
        n -> padding = va_arg (*ap, int);
        n -> vmin = va_arg (*ap, long);
        n -> vmax = va_arg (*ap, long);
    }
    return (char *) n;
}
static char * copy_hex (arg)
char * arg;
{
    HEX * n = (HEX *) malloc (sizeof (HEX));
    if (n) *n = *((HEX *) arg);
    return (char *) n;
}
static void free_hex (arg)
char * arg;
{
    free (arg);
}


static int fcheck_hex (f, arg)
FIELD * f;
char * arg;
{
    HEX * n = (HEX *) arg;
    int padding = n -> padding;
```

**Screen 11-19. Creating TYPE_HEX with Padding and Range Arguments**

```
        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_fieldtype (fcheck_hex, ccheck_hex);
    set_fieldtype_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. So there is

no need for functions (such as `fcheck_hex`) that use these values to check that they are not NULL.

If successful, function `set_fieldtype_arg` returns `E_OK.` If an error occurs, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | field type, *make_arg*, *copy_arg*, or *free_arg* is `NULL` |

## Supporting Next and Previous Choice Functions

Some field types comprise a set of values from which your user chooses (enters) one. The following functions support those types that have a set of choices.

### SYNOPSIS

```
typedef char * (* PTF_charP) ();

int set_fieldtype_choice (type, next_choice, prev_choice)
FIELDTYPE * type;
PTF_int next_choice;
PTF_int prev_choice;

int next_choice(f,arg);
FIELD * f;
char * arg;

int prev_choice(f,arg);
FIELD * f;
char * arg;
```

These functions enable the ETI form driver to support the `REQ_NEXT_CHOICE` and `REQ_PREV_CHOICE` requests mentioned in “Form Driver Processing” on page 11-40.

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.

Screen 11-20 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.

```
static int next_hex (f, arg)
FIELD * f;
char * arg;
{
    HEX * n = (HEX *) arg;
    long v = n -> vmin;
    char buf[80];
    char * x = field_buffer (f, 0);

    while (*x && isblank (*x)) ++x;

    if (*x)
    {
        v = strtol (x, (char **) 0, 16);
        if (v >= n -> vmin && v < n -> vmax)
            ++v;
    }
    sprintf (buf, “%.*lx”, n -> padding, v);
    set_field_buffer (f, 0, buf);
    return TRUE;
}
static int prev_hex (f, arg)
FIELD * f;
char * arg;
{
    HEX * n = (HEX *) arg;
    long v = n -> vmax;
    char buf[80];
    char * x = field_buffer (f, 0);

    while (*x && isblank (*x)) ++x;

    if (*x)
    {
        v = strtol (x, (char **) 0, 16);
        if (v > n -> vmin && v <= n -> vmax)
            --v;
    }
    sprintf (buf, “%.*lx”, v -> padding, v);
    set_field_buffer (f, 0, buf);
    return TRUE;
}

        /* associate previous and next choice functions */
set_fieldtype_choice (TYPE_HEX, next_hex, prev_hex);
```

**Screen 11-20.  Creating a Next Choice Function for a Field Type**

If given a blank field, your functions *next_choice* and *prev_choice* should, of course, do something reasonable, such as setting the field to the first or last value of the type.

If function `set_fieldtype_choice` encounters an error, it returns one of the following:

| | |
|---|---|
| `E_SYSTEM_ERROR` | system error |
| `E_BAD_ARGUMENT` | field type, `next_choice`, or `prev_choice` is NULL |

# 12
# Other ETI Routines

# 12 Other ETI Routines

## Introduction

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(3curses)** manual pages before you try to use the following ETI features.

## Routines for Drawing Lines and Other Graphics

Many terminals have an alternate character set for drawing simple graphics (or glyphs or graphic symbols). You can use this character set in ETI programs. ETI uses the same names for glyphs as the VT100 line drawing character set.

To use the alternate character set in an ETI program, you pass a set of variables whose names begin with ACS_ to the ETI routine waddch or a related routine. For example, ACS_ULCORNER is the variable for the upper left corner glyph. If a terminal has a line drawing character for this glyph, ACS_ULCORNER's value is the terminal's character for that glyph OR'd ( | ) with the bit-mask A_ALTCHARSET. If no line drawing character is available for that glyph, a standard ASCII character that approximates the glyph is stored in its place. For example, the default character for ACS_HLINE, a horizontal line, is a - (minus sign). When a close approximation is not available, a + (plus sign) is used. All the standard ACS_ names and their defaults are listed on the **curses(3curses)** manual pages.

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. 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 the **`curses(3curses)`** pages in this guide.

## 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`, and `slk_restore`.

When you use soft labels in an ETI program, you have to call the routine `slk_init` before `initscr`. This sets an internal flag for `initscr` to look at that 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 one 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 `slk_refresh` 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`.

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(3curses)`** manual pages for more information about the routines for using soft labels.

## Working with More Than One Terminal

An ETI program can produce output on more than one terminal at the same time. This is useful for single process programs that access a common database, such as multi-player games.

Writing programs that output to multiple terminals is a difficult business, and the ETI library does not solve all the problems you might encounter. For instance, the programs—not the library routines—must determine the file name of each terminal line, and what kind of terminal is on each of those lines. The standard method, checking `$TERM` in the environment, does not work, because each process can only examine its own environment.

Another problem you might face is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. However, a program trying to take over another terminal cannot just shut off whatever program is currently running on that line. (Usually, security reasons would also make this inappropriate. But, for some applications, such as an inter-terminal communication program, or a program that takes over unused terminal lines, it would be appropriate.) A typical solution to this problem requires each user logged in on a line to run a program that notifies a master program that the user is interested in joining the master program and tells it the notification program's process ID, the name of the tty line, and the type of terminal being used. Then the program goes to sleep until the master program finishes. When done, the master program wakes up the notification program and all programs exit.

An ETI program handles multiple terminals by always having a current terminal. All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When it wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary ETI routines.

References to terminals in an ETI program have the type `SCREEN*`. A new terminal is initialized by calling `newterm(`*type, outfd, infd*`)`. `newterm` returns a screen reference to the terminal being set up. *type* is a character string, naming the kind of terminal being used. *outfd* is a **`stdio(3S)`** file pointer (`FILE*`) used for output to the terminal and *infd* a file pointer for input from the terminal. This call replaces the normal call to `initscr`, which calls `newterm(getenv(”TERM”), stdout, stdin)`.

To change the current terminal, call `set_term`*(sp)* where *sp* is the screen reference to be made current. `set_term` returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with `newterm`. Options such as `cbreak` and `noecho` must be set separately for each terminal. The functions `endwin` and `refresh` must be called separately for each terminal. Screen 12-1 shows a typical scenario to output a message to several terminals.

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

**Screen 12-1. Sending a Message to Several Terminals**

See the **`two`** program in Appendix D for a more complete example.

# 13
# terminfo

# 13 terminfo

## Introduction

This chapter explains how to use the **`terminfo`** database and the **`terminfo`** routines to write terminal-independent screen management programs on the UNIX system. Other support tools are also described.

The purpose of this chapter is to explain how to write screen management programs as quickly as possible. Therefore, this chapter does not attempt to cover every detail. Use this chapter to get familiar with the way these routines work, then use the manual pages for more information.

## Organization of This Chapter

This chapter has the following sections:

- "What Is terminfo?" on page 13-1 introduces you to the **`terminfo`** routines and the **`terminfo`** database.
- "Working with terminfo Routines" on page 13-2 describes how the **`terminfo`** routines access and manipulate data in the **`terminfo`** database.
- "Working with the terminfo Database" on page 13-6 describes the **`terminfo`** database, related support tools, and their relationship to the **`curses`** library.

## What Is terminfo?

**`terminfo`** refers to both of the following:

- It is a group of routines within the **`curses`** library that handles certain terminal capabilities. You can use these **`terminfo`** routines to write a filter or program the function keys, if your terminal has programmable keys. Shell programmers can use the command **`tput(1)`** to perform many of the manipulations provided by these routines.
- It is a database containing the descriptions of many terminals that can be used with **`curses`** programs. These descriptions specify the capabilities of a terminal and the way it performs various operations—for example, how many lines and columns it has and how its control characters are interpreted.

Each terminal description in the database is a separate, compiled file. You use the source code that **`terminfo(4)`** describes to create these files and the command **`tic(1M)`** to compile them.

The compiled files are normally located in the directories **`/usr/share/lib/terminfo/?`**. These directories have single character names, each of which is the first character in the name of a terminal. For example, an entry for the AT&T Teletype 5425 is normally located in the file **`/usr/share/lib/terminfo/a/att5425`**.

Here's a simple shell script that uses the **`terminfo`** database.

```
#   Clear the screen and show the 0,0 position.

tput clear
tput cup 0 0       #   or tput home
echo “<- this is 0 0”

#   Show line 5, column 10.

tput cup 5 10
echo “<- this is 5 10”
```

**Screen 13-1.  A Shell Script Using terminfo Routines**

## Working with terminfo Routines

Some programs need to use lower level routines (that is, 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 stand out 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: the higher level **`curses`** routines make your program more portable to other UNIX systems and to a wider class of terminals.

**NOTE**

You are discouraged from using **terminfo** routines except for the purposes noted, because **curses** routines take care of all the glitches present in physical terminals. When you use the **terminfo** routines, you must deal with the glitches yourself. Also, these routines may change and be incompatible with previous releases.

## What Every terminfo Program Needs

A **terminfo** program typically includes the header files and routines shown in Screen 13-2.

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

**Screen 13-2. Typical Framework of a terminfo Program**

The header files **curses.h** and **term.h** are required because they contain the definitions of the strings, numbers, and flags used by the **terminfo** routines. `setupterm` takes care of initialization. Passing this routine the values `(char*)0, 1,` and `(int*)0` invokes reasonable defaults. If `setupterm` can't figure out what kind of terminal you are on, it prints an error message and exits. `reset_shell_mode` performs functions similar to `endwin` and should be called before a **terminfo** program exits.

A global variable like `clear_screen` is defined by the call to `setupterm`. It can be output using the **terminfo** routines `putp` or `tputs`, which gives a user more control. This string should not be directly output to the terminal using the C library routine **printf(3S)**, because it contains padding information. A program that directly outputs strings will fail on terminals that require padding or that use the `xon/xoff` flow control protocol.

At the **terminfo** level, the higher level routines like `addch` and `getch` are not available. It is up to you to output whatever is needed. For a list of capabilities and a description of what they do, see **terminfo(4)**; see **curses(3curses)** 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 “Compiling an ETI Program” on page 6-4 and “Running an ETI Program” on page 6-4 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 Appendix D) that does not use the higher level **`curses`** routines. **`termhl`** can be used as a filter. It includes the strings to enter bold and underline mode and to turn off all attributes.

```
/*
 * A terminfo level version of the highlight program.
 */

#include <curses.h>
#include <term.h>

int ulmode = 0;  /* Currently underlining */

main(argc, argv)
  int argc;
  char **argv;
{
   FILE *fd;
   int c, c2;
   int outch();

   if (argc > 2)
   {
      fprintf(stderr, “Usage: termhl [file]\n”);
      exit(1);
   }

   if (argc == 2)
   {
      fd = fopen(argv[1], “r”);
      if (fd == NULL)
      {
         perror(argv[1]);
         exit(2);
      }
   }
   else
   {
      fd = stdin;
   }
   setupterm((char*)0, 1, (int*)0);
```

**Screen 13-3.  Example of terminfo Program**

```
 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 “Specify Capabilities” on page 13-8). `tputs` generates enough pad characters to delay for the appropriate time.

`tputs` has three parameters. The first parameter is the string capability to be output. The second is the number of lines affected by the capability. (Some capabilities may require padding that depends on the number of lines affected. For example, `insert_line` may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention *affcnt* is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since *affcnt* is multiplied by the amount of time per item, and anything multiplied by 0 is 0.) The third parameter is a routine to be called with each character.

For many simple programs, *affcnt* is always 1 and *outc* always calls `putchar`. For these programs, the routine `putp`(*cap*) is a convenient abbreviation. **`termhl`** could be simplified by using `putp`.

Now to understand why you should use the **`curses`** level routines instead of **`terminfo`** level routines whenever possible, note the special check for the `underline_char` capability in this sample program. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. **`termhl`** keeps track of the current mode, and if the current character is supposed to be underlined, outputs `underline_char`, if necessary. Low level details such as this are precisely why the **`curses`** level is recommended over the **`terminfo`** level. **`curses`** takes care of terminals with different methods of underlining and other terminal functions. Programs at the **`terminfo`** level must handle such details themselves.

**`termhl`** was written to illustrate a typical use of the **`terminfo`** routines. It is more complex than it need be in order to illustrate some properties of **`terminfo`** programs. The routine `vidattr` (see **`curses(3curses)`**) could have been used instead of directly outputting `enter_bold_mode`, `enter_underline_mode`, and `exit_attribute_mode`. In fact, the program would be more robust if it did, since there are several ways to change video attribute modes.

## Working with the terminfo Database

The **`terminfo`** database describes the many terminals with which **`curses`** programs, as well as some UNIX system tools, like **`vi(1)`**, 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 contained 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 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(5)`** describes mode indicators in greater detail.

## Learn About the Capabilities

After you complete the string of terminal names for your description, you have to learn about the terminal's capabilities so that you can properly describe them. To learn about the capabilities your terminal has, you should do the following:

- See the owner's manual for your terminal. It should have information about the capabilities available and the character strings that make up the sequence transmitted from the keyboard for each capability.

- Test the keys on your terminal to see what they transmit, if this information is not available in the manual. You can test the keys in one of the following ways. Type:

  **`stty -echo; cat -vu`**
  *Type in the keys you want to test;*
  *for example, see what right arrow* `(<Right-Arrow>) transmits.`
  `<CR>`
  `<CTRL-D>`
  **`stty echo`**

  or

  **`cat >dev/null`**
  *Type in the escape sequences you want to test;*
  *for example, see what* `\E[H` *transmits.*
  `<CTRL-D>`

- The first line in each of these testing methods sets up the terminal to carry out the tests. The `<CTRL-D>` helps return the terminal to its normal settings.

- See the **`terminfo(4)`** manual page. It lists all the capability names you have to use in a terminal description. “Specify Capabilities” on page 13-8 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(4)`** 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 `ESCAPE-C`. |
| `\n` | These characters provide a `<NL>` character sequence. |
| `\l` | These characters provide a linefeed character sequence. |
| `\r` | These characters provide a return character sequence. |
| `\t` | These characters provide a tab character sequence. |
| `\b` | These characters provide a backspace character sequence. |
| `\f` | These characters provide a form-feed character sequence. |
| `\s` | These characters provide a space character sequence. |
| `\`*nnn* | This is a character whose three-digit octal is *nnn*, where *nnn* can be one to three digits. |
| `$< >` | These symbols are used to show a delay in milliseconds. The desired length of delay is enclosed inside the "less than/greater than" symbols (`< >`). The amount of delay may be a whole number, a numeric value to one decimal place (tenths), or either form followed by an asterisk (*). The * shows that the delay will be proportional to the number of lines affected by the operation. For example, a 20-millisecond delay per line would appear as `$<20*>`. See the **`terminfo(4)`** manual page for more information about delays and padding. |

Sometimes, it may be necessary to comment out a capability so that the terminal ignores this particular field. This is done by placing a period ( `.` ) in front of the abbreviated name for the capability. For example, if you would like to comment out the beeping capability, the description entry would appear as

```
.bel=^G,
```

With this background information about specifying capabilities, let's add the capability string to our description of `myterm`. We'll consider basic, screen-oriented, keyboard-entered, and parameter string capabilities.

## Basic Capabilities

Some capabilities common to most terminals are bells, columns, lines on the screen, and overstriking of characters, if necessary. Suppose our fictitious terminal has these and a few other capabilities, as listed below. Note that the list gives the abbreviated **`terminfo`** name for each capability in the parentheses following the capability description:

- 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 “Name the Terminal” on page 13-7) and the capability descriptions that we now have, we get the following general **`terminfo`** database entry:

```
myterm|mytm|mine|fancy|terminal|My FANCY terminal,
    am, bel=^G, cols#80, lines#30, xon,
```

## Screen-oriented Capabilities

Screen-oriented capabilities manipulate the contents of a screen. Our example terminal `myterm` has the following screen-oriented capabilities. Again, the abbreviated command associated with the given capability is shown in parentheses.

- A <CR> is a CTRL-M (**`cr`**).
- A cursor up one line motion is a CTRL-K (**`cuu1`**).
- A cursor down one line motion is a CTRL-J (**`cud1`**).
- Moving the cursor to the left one space is a CTRL-H (**`cub1`**).
- Moving the cursor to the right one space is a CTRL-L (**`cuf1`**).
- Entering reverse video mode is an ESCAPE-D (**`smso`**).
- Exiting reverse video mode is an ESCAPE-Z (**`rmso`**).
- A clear to the end of a line sequence is an ESCAPE-K and should have a 3-millisecond delay (**`el`**).
- A terminal scrolls when receiving a <NL> at the bottom of a page (**`ind`**).

The revised terminal description for `myterm` including these screen-oriented capabilities follows:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
    am, bel=^G, cols#80, lines#30, xon,
    cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
    smso=\ED, rmso=\EZ, el=\EK$<3>, ind=\n,
```

### Keyboard-entered Capabilities

Keyboard-entered capabilities are sequences generated when a key is typed on a terminal keyboard. Most terminals have, at least, a few special keys on their keyboard, such as arrow keys and the backspace key. Our example terminal has several of these keys whose sequences are, as follows:

- The backspace key generates a CTRL-H (**kbs**).
- The up arrow key generates an ESCAPE-[ A (**kcuu1**).
- The down arrow key generates an ESCAPE-[ B (**kcud1**).
- The right arrow key generates an ESCAPE-[ C (**kcuf1**).
- The left arrow key generates an ESCAPE-[ D (**kcub1**).
- The home key generates an ESCAPE-[ H (**khome**).

Adding this new information to our database entry for `myterm` produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
    am, bel=^G, cols#80, lines#30, xon,
    cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
    smso=\ED, rmso=\EZ, el=\EK$<3>, ind=
    kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
    kcub1=\E[D, khome=\E[H,
```

### Parameter String Capabilities

Parameter string capabilities are capabilities that can take parameters — for example, those used to position a cursor on a screen or turn on a combination of video modes. To address a cursor, the `cup` capability is used and is passed two parameters: the row and column to address. String capabilities, such as `cup` and set attributes (`sgr`) capabilities, are passed arguments in a **`terminfo`** program by the `tparm` routine.

The arguments to string capabilities are manipulated with special '%' sequences similar to those found in a **`printf(3S)`** statement. In addition, many of the features found on a simple stack-based RPN calculator are available. `cup`, as noted above, takes two arguments: the row and column. `sgr`, takes nine arguments, one for each of the nine video attributes. See **`terminfo(4)`** for the list and order of the attributes and further examples of `sgr`.

Our fancy terminal's cursor position sequence requires a row and column to be output as numbers separated by a semicolon, preceded by `ESCAPE-[` and followed with `H`. The coordinate numbers are 1-based rather than 0-based. Thus, to move to row 5, column 18, from (0,0), the sequence `'ESCAPE-[ 6 ; 19 H'` would be output.

Integer arguments are pushed onto the stack with a '`%p`' sequence followed by the argument number, such as '`%p2`' to push the second argument. A shorthand sequence to increment the first two arguments is '`%i`'. To output the top number on the stack as a decimal, a '`%d`' sequence is used, exactly as in `printf`.

Our terminal's `cup` sequence is built up as follows:

| cup= | Meaning |
|---|---|
| \E[ | output `ESCAPE-[` |
| %i | increment the two arguments |
| %p1 | push the 1st argument (the row) onto the stack |
| %d | output the row as a decimal |
| ; | output a semi-colon |
| %p2 | push the 2nd argument (the column) onto the stack |
| %d | output the column as a decimal |
| H | output the trailing letter |

or

```
cup=\E[%i%p1%d;%p2%dH,
```

Adding this new information to our database entry for `myterm` produces:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
    am, bel=^G, cols#80, lines#30, xon,
    cr=^M, cuu1=^K, cud1=^J, cub1=^H, cuf1=^L,
    smso=\ED, rmso=\EZ, el=\EK$<3>, ind=
    kbs=^H, kcuu1=\E[A, kcud1=\E[B, kcuf1=\E[C,
    kcub1=\E[D, khome=\E[H,
    cup=\E[%i%p1%d;%p2%dH,
```

See **`terminfo(4)`** for more information about parameter string capabilities.

## Compile the Description

The **`terminfo`** database entries are compiled using the **`tic`** compiler. This compiler translates **`terminfo`** database entries from the source format into the compiled format.

The source file for the description is usually in a file suffixed with **`.ti`**. For example, the description of `myterm` would be in a source file named **`myterm.ti`**. The compiled description of `myterm` would usually be placed in **`/usr/share/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/share/lib/terminfo`**. The general format for the **`tic`** compiler is as follows:

```
tic [-v] [-c] file
```

The **`-v`** option causes the compiler to trace its actions and output information about its progress. The **`-c`** option causes a check for errors; it may be combined with the **`-v`** option. *file* shows what file is to be compiled. If you want to compile more than one file at the same time, you have to first use **`cat(1)`** to join them together. The following command line shows how to compile the **`terminfo`** source file for our fictitious terminal:

```
tic -v myterm.ti  RETURN
```

(The trace information appears as the compilation proceeds.)

Refer to the **`tic(1M)`** 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(1)`**) a large file (over 100 lines) at 9600 baud (if possible), and deleting about 15 lines from the middle of the screen. Type `u` (undo) several times quickly. If the terminal messes up, then more padding is usually required. A similar test can be used for inserting a character.

Third, you can use the **`tput(1)`** 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 **`-T`***type* option. Usually, this option is not necessary because the default terminal name is taken from the environment variable `$TERM`. The *capname* field is used to show what capability to output from the **`terminfo`** database.

The following command line shows how to output the "clear screen" character sequence for the terminal being used:

```
tput clear
```

(The screen is cleared.)

The following command line shows how to output the number of columns for the terminal being used:

```
tput cols
```

(The number of columns used by the terminal appears here.)

The **`tput(1)`** manual page contains more information on the usage and possible messages associated with this command.

## Comparing or Printing terminfo Descriptions

Sometime you may want to compare two terminal descriptions or quickly look at a description without going to the **`terminfo`** source directory. The **`infocmp(1M)`** command was designed to help you with both of these tasks. Compare two descriptions of the same terminal; for example,

```
mkdir /tmp/old /tmp/new
TERMINFO=/tmp/old tic old5420.ti
TERMINFO=/tmp/new tic new5420.ti
infocmp -A /tmp/old -B /tmp/new -d 5420 5420
```

compares the old and new 5420 entries.

To print out the **`terminfo`** source for the 5420, type

```
infocmp -I 5420
```

## Converting a termcap Description to a terminfo Description

**CAUTION**

The **`terminfo`** database is designed to take the place of the `termcap` database. Because of the many programs and processes that have been written with and for the `termcap` database, it is not feasible to do a complete cutover at one time. Any conversion from `termcap` to **`terminfo`** requires some experience with both databases. All entries into the databases should be handled with extreme caution. These files are important to the operation of your terminal.

The **`captoinfo(1M)`** command converts `termcap` descriptions to **`terminfo(4)`** descriptions. When a file is passed to **`captoinfo`**, it looks for `termcap` descriptions and writes the equivalent **`terminfo`** descriptions on the standard output. For example,

```
captoinfo /usr/share/lib/termcap
```

converts the file **`/usr/share/lib/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. This is recommended because the **`terminfo`** entry will be more complete, descriptive, and accurate than the `termcap` entry possibly could be.

If you have been using cursor optimization programs with the **`-ltermcap`** or **`-ltermlib`** option in the **`cc`** command line, those programs will still be functional. However, these options should be replaced with the **`-lcurses`** option.

# A
# Programming Tips and Known Problems

## Programming Tips

### Internationalization Support

FMLI accepts as input any character from a standard 7- or 8-bit character set. This means that descriptor and variable values and application-specific command names may be coded in a language other than English, provided the language implementation employs a standard 8-bit code set. It also means that users may enter input in a form, or edit the text in a text frame, in any such language. Note, however, that the built-in utilities **`fmlexpr(1F)`**, **`fmlgrep(1F)`**, and **`regex(1F)`** do not support regular expression matching for non-ASCII character sets, and that FMLI error messages are always displayed in English.

FMLI uses the **`setlocale(3C)`** function to examine the user's environment for a current *locale*—a collection of information that describes conventions appropriate to some nationality, culture, and language. This information is stored in databases that specify how to sort or classify characters, for instance, according to these conventions. If such databases exist on a user's system, they are accessed through the `LANG` variable in the user's environment. An application coded for a German locale, then, should instruct users to set the `LANG` environment variable to `de`[utsche]; character classification, sorting, and so on will be done in the appropriate way. For details on this mechanism, see the **`setlocale(3C)`** and **`environ(5)`** manual pages.

### Building Trusted FMLI Applications

This section gives tips on how to build FMLI applications that prevent unauthorized users from circumventing access controls or mechanisms that protect sensitive system operations.

#### Access to External Executables

An FMLI application may allow users to access UNIX system executables by escaping to the shell, either with the `!` shell escape or the **`unix-system`** command, or by selecting a menu item or SLK for which the action coded invokes an external executable. It is your responsibility to make sure these means of access are not abused.

Both means of escaping to the shell can be disabled if necessary. When the `nobang` descriptor is defined and evaluates to TRUE, users cannot use the `!` shell escape to invoke external executables. Nor will they be able to invoke the **`open`** command from the command line. That means the user cannot, for instance, create a menu that invokes an external executable and open it to gain access to the shell. You can disable the **`unix-system`** command as described in “The Commands File” on page 4-12. Keep in mind that disabling the command will also make it unavailable to your own FMLI scripts.

FMLI does not try to second-guess your application by setting shell variables such as `PATH` or `IFS`. It leaves it to you to insure that the execution environment is correct, and that the correct executable will be found. Similarly, it is your responsibility to make sure that any commands you add or redefine in the commands file perform as expected and do not violate security principles. For discussions, see Chapter 4.

## Interruptible Commands

By default, FMLI commands and executables invoked through FMLI (except those running under a full-screen mode executable) are not interruptible. If you enable interrupts for some or all parts of your application, you must insure that not completing the interrupted action will not compromise the system and that `oninterrupt` is defined such that the appropriate recovery actions are taken. The risks here include leaving sensitive data files or incomplete or inconsistent data on the system after the unfinished operation. You need to weigh these risks against the “friendliness” of allowing interrupts. At the very least, interrupts should be enabled at the lowest possible level, rather than for the whole application. The `oninterrupt` in effect should always be the appropriate one. It, too, should be defined at the lowest level of the hierarchy. For a discussion, see “Interrupt Signal Handling” on page 2-39.

## Variables

FMLI uses four distinct types of variables, each of which can produce unexpected results if handled improperly.

UNIX system shell environment variables are inherited from the invocation environment and can be modified with the **`set`** and **`unset`** built-in utilities. The effects of any modification are visible to child processes of the application.

File-based variables are stored in and read from the file that is part of their name, so that multiple applications or sessions can share data.

So-called local variables are local to FMLI but global within the application. Any line that references `$foo`, for example, will reference the same value; changing the variable's value in one frame changes its value in all frames. For this reason, extreme care should be taken to avoid name collision when developing applications, especially when there is more than one programmer involved. For this reason, too, you should be sure to implement validation tests with appropriate criteria when using FMLI's mechanisms for validating user input. In particular, if you set a variable based on user input in one frame and reference it in another, the data may be valid at the time of entry but not at the time of use. For a discussion and example, see “Validation of Form Fields” on page A-4.

Built-in variables are read-only variables that are visible only within the FMLI session. Some are maintained on a per-frame basis and have different values in each frame. Others are visible throughout the application.

There are two ways to dereference a variable. `$`*VAR* parses the variable once. `$!`*VAR* reparses the variable as long as it contains special characters. That is, if you set `VAR` to `` `pwd` ``, `$VAR` is `` `pwd` ``, whereas `$!VAR` may be **`/home/chris`**. You should never use the `$!` notation when referencing built-in variables because it is impossible to guard against users entering special characters in fields. Note that if the initialization file descriptor `use_incorrect_pre_4.0_behavior` is set to TRUE, the first form implies the second. For discussions, see "Variables" on page 2-6.

**NOTE**

A significant amount of FMLI code was written to implement software that is now obsolete. This code should not be used by other applications. To avoid unexpected results in this regard, the UNIX shell environment variable `VMFMLI` should not be passed to the FMLI environment. Better still, the variable can be set to FALSE. Since this variable is tested only when FMLI is invoked, changes made to it later by the user will not affect FMLI's behavior.

## Frequency of Evaluation Type Casts

By default, FMLI determines how often descriptors are evaluated. You can use the `const` type cast to make sure that a descriptor is evaluated only once, no matter how many times it is referenced, or the `vary` type cast to make sure that a descriptor is evaluated whenever it is referenced. Use these casts with care. In particular, do not use `const` if the data could become out of date, that is, unless you know that the descriptor value will be the same no matter how many times it is referenced, or are certain that the first value must be retained (a startup-time value, for example). For a discussion, see "Descriptor Evaluation" on page 2-10.

## Co-processing

Co-processes for trusted applications should use named pipes created by the application with the appropriate permissions; the default pipes created by FMLI are readable and writable by everyone. Handshaking can also be used to enhance security. For a discussion and examples, see "Other Useful Examples" on page 3-61.

## FACE-specific Code

There is a significant amount of code in FMLI that was written to implement FACE. This code should not be used by other applications. To avoid unexpected results in this regard, the UNIX shell environment variable `VMFMLI` should not be passed to the FMLI environment. Better still, the variable can be set to FALSE. Because it is tested only when FMLI is invoked, changes made to it by the user after that will not affect FMLI's behavior.

Although the executable `facesuspend` is provided only with the FACE product, access to it may give the knowledgeable user the ability to return to an FMLI application before completing a task begun in a full-screen window. Access to this executable should be restricted. Alternatively, appropriate validation should be performed to insure that the task begun in the full-screen window was completed.

## Validation of Form Fields

Developers should be aware that information stored in FMLI variables via the **`set`** utility or the argument passing mechanism (`$ARG`*n*) may not be valid at the time it is used even if it was validated at the time it was set. This can occur when variables set from data in one frame are used in the processing activity of another frame. If the use of a variable containing invalid data could seriously corrupt or compromise the system, it must be re-validated at the time it is used.

Developer-set variables are known to all frames in an FMLI session—there is no *frame scoping* of variables, no way to make a variable known only to the frame it is set in. This results in the classic programming issues around global variables.

Here are two scenarios that can result in the value of a variable no longer being valid.

### Scenario 1

The `done` descriptor of `Form.1` sets a variable **`set -l`** `FOO=$F1`, the value of field one, and opens `Form.1a.` The user enters data in `Form.1` and presses the `SAVE` SLK; `Form.1a` opens and becomes the current frame. The user now has a change of mind, navigates back to `Form.1`, and enters a new value in field one. If, instead of pressing the `SAVE` SLK for `Form.1` again, the user navigates to `Form.1a`, when the user saves `Form.1a` it will not know the value in `Form.1` has changed and any action in `Form.1a` based on the value of `FOO` will be different from what the user expects. The user's error of not pressing `SAVE` after changing `Form.1` will not be detected.

### Scenario 2

The done descriptor of `Form.1` opens `Form.1a` passing the value of `$F1` as the first argument (as in `open Form.1a $F1`). Assume this value is a user ID that `Form.1` validated. Now the user navigates to another menu and deletes the user, then navigates back to `Form.1a`. Now the value of `ARG1` is not a valid user ID even though `Form.1` validated it. `Form.1a` must re-validate the value before doing anything based on it.

## Commands

- If an FMLI application initiates a call to a UNIX system command (for example, ``action=`unix_command`nop``) and the `interrupt` descriptor evaluates to FALSE for that `action` descriptor (see “Interrupt Signal Handling” on page 2-39), users will not be able to do other tasks until the command completes even if the command could be interrupted. If the command takes a considerable amount of time to execute, the application writer may want the command to execute in the background.

Since FMLI does not recognize the shell background symbol `&`, the **`shell`** built-in command must be used, for instance,

```
action=`shell "unix_command > /dev/null &"`nop
```

If you want the user to continue to be able to interact with the application while the background job is running, the output of an executable run by **`shell`** in the background must be redirected: to a file if you want to save the output, or to **`/dev/null`** if you don't want to save it (or if there is no output); otherwise your application may appear to be hung until the background job finishes processing. The application writer may also wish to explore the co-processing facility **`coproc(1F)`** which establishes a pipe between FMLI and another independent UNIX system process.

- When an FMLI command is disabled in the commands file, as in

  ```
  name=update
  action=nop
  ```

  this disables it throughout the interface. There is no way to remove it from the `Command Menu` and still leave it available for use in the application code itself.

## Co-processing Functions

- When writing programs to use as co-processes, the following tips may be useful. If the co-process program is written in C language, be sure to flush output after writing to the pipe. (Currently, **`awk(1)`** and **`sed(1)`** cannot be used in a co-process program because they do not flush after lines of output.) Shell scripts are well-mannered, but slow. C language is recommended. If possible, use the default *send_string*, *rpath*, and *wpath*. In most cases, *expect_string* has to be specified. This, of course, depends on the co-process.

## Forms

- Choices for a form field can be specified using the `rmenu` descriptor. If the value of `rmenu` is a list of items enclosed in brackets, there must be at least one whitespace character that separates the brackets from the item list: `rmenu={ "item 1" "item 2" "item 3" }`.
- If a definition of the `rmenu` descriptor degenerates to an empty list, `rmenu={}`, the value of `choicemsg` is displayed—your definition if you have defined one, or the FMLI default message `There are no choices available`. If you define `choicemsg` and there might not be any choices, be sure the message is appropriate to the "empty list" case.
- There must be at least one active field visible in a form. If you open a form with all fields defined as inactive, or `show=FALSE`, FMLI does not display the frame.

- Field *n* in a form frame cannot reference field *m*, where *m* is greater than *n*, and field *m* does not have a `value` descriptor defined. That is, you cannot reference the value of a field that is defined later in the form definition file because that field may not have been evaluated at the time you reference it.
- If a second or subsequent page of a form is defined to be larger than can be displayed on the terminal being used, it is not displayed at all (for example, if `rows=25` is defined, and the terminal being used only has 24 rows available for display).

## Menus

- The precise rules for how rows and columns are determined in menus are given in the following table. This table should only be needed in exceptional cases (for example, when a developer has coded "unreasonable" values for the `rows` and `columns` descriptors in a menu definition file). In general, the number of columns in a menu is determined before the number of rows, and columns specified with the `columns` descriptor takes precedence if there is a conflict with the number of rows requested. The number of rows is usually the minimum of the three variables *aR* (available rows), *sR* (specified rows), and *nR* (needed rows).

  The table entries for the two cases when `columns` is specified and `description` is not specified imply that menu items are truncated to fit in the column size determined from *sC*. Thus, the `columns` descriptor should not be specified for menus that are dynamically generated, when there is no way to guarantee that such a menu will not have truncated items.

| Descriptors set? | | | step 1 | step 2 | step 3 | |
|---|---|---|---|---|---|---|
| `d` | `r` | `c` | *pC* | *pR* | *uC* | *uR* |
| yes | no | no | not needed | not needed | 1 | min(10,*nR*) |
| yes | no | yes | not needed | not needed | 1* | min(10,*nR*) |
| yes | yes | no | not needed | not needed | 1 | min(*aR*,*nR*,*sR*) |
| yes | yes | yes | not needed | not needed | 1* | min(*aR*,*nR*,*sR*) |
| no | no | no | —** | —** | —** | —** |
| no | no | yes | if *sC* > *mC*, 1; otherwise, *sC**** | ((*tI*-1) mod *pC*)+1 | if *pR* > *aR*, 1: otherwise, *pC* | if *pR* > *aR*, min(*aR*,10); otherwise, *pR* |
| no | yes | no | ((*tI*-1) mod *pR*)+1† | min(*aR*,*nR*,*sR*)† | if *pC* > *fC*, 1; otherwise, *pC* | *pR* |
| no | yes | yes | if *sC* > *mC*, 1; otherwise, *sC**** | ((*tI*-1) mod *pC*)+1 | if *pR* > *aR*, 1; otherwise, *pC* | if *pR* > *aR*, min(*aR*,*sR*); otherwise, *pR*†† |

Footnotes

| | |
|---|---|
| * | `columns` descriptor is ignored |
| ** | the algorithm attempts to open a menu with a 3:1 aspect ratio of width to height |
| *** | menu items are truncated if they are too long to fit; equal-width columns are kept after truncation |
| † | step 1 and step 2 are reversed for this case (*pR* must be computed first) |
| †† | `rows` descriptor is ignored |

Legend

| | |
|---|---|
| `d` | `description` descriptor |
| `r` | `rows` descriptor |
| `c` | `columns` descriptor |
| *sR* | (specified rows) the value coded with the `rows` descriptor |
| *aR* | (available rows) the number of rows that frames can occupy on the terminal screen |
| *nR* | (needed rows) the number or rows needed to open the menu—for single-column menus this equals *tI* |
| *pR* | (probable rows) the number of rows needed to open the menu, as determined from the first (preliminary) calculations |
| *uR* | (used rows) the number of rows used to open the menu, after all steps are done |
| *tI* | (total items) the total number of menu items for the menu (the number of `menu` descriptors) |
| *sC* | (specified columns) the value coded with the `columns` descriptor |
| *fC* | (*fittable columns*) the number of columns that can fit on the screen, given the screen width and the length of the longest menu item; equals (*screenWidth*-2) mod (*maxItemWidth*+1); this is a *maximum value*—the maximum fittable columns |
| *mC* | (max columns) the maximum number of columns that could fit on the screen if each column were only 1 character wide; equals (*screenWidth*-3) mod 2 |
| *pC* | (probable columns) the number of columns needed to open the menu, as determined from the first (preliminary) calculations |
| *uC* | (used columns) the number of columns used to open the menu, after all steps are done |

## Text

- The SCROLL-DOWN key displays the complete final page of a text frame, even if much of it was already visible. The SCROLL-UP key displays the entire first page of a text object, even if most of it was already visible. The action of SCROLL-DOWN might be a surprise to users if they are not also aware that the scroll-down icon has disappeared, signaling that they are at the end of the text.

## Backquoted Expressions

- Backquoted expressions that appear on a line by themselves are evaluated before any descriptors are parsed. That is, they are evaluated before the frame is fully current. Thus, the following can occur:
  - if a stand-alone backquoted expression produces output to the message line, it may appear before the frame being parsed is posted. This delay may or may not be significant and depends on the complexity of the frame definition file.
  - **`message -f`** statements in stand-alone backquoted expressions are ignored.
  - the built-in function `getfrm`, if used in a stand-alone backquoted expression, may be parsed before the frame ID it is supposed to return is available.
- If a command run in a backquoted expression changes the **`stty(1)`** setting, the FMLI session may be corrupted. Frames may not display correctly and the command line may not function (the latter occurs if RETURN is mapped to LINEFEED or to RETURN LINEFEED).
- If a daemon process is started via a shell script that FMLI code invokes in a backquoted expression, FMLI waits for this process until the UNIX system clears up zombies. While waiting, FMLI appears to be locked because the backquoted command that was `exec`'d created a child whose `stdout` is still connected to FMLI via a pipe. When the command becomes a zombie, FMLI continues reading the pipe that the (daemon) child still has open. FMLI does not know if its grandchildren are going to be daemons or if they are going to write to the pipe. To preserve the ability of grandchildren to output to FMLI, the following fix must be placed in the script executed by the backquoted expression, to redirect the `stdout` of the daemon:

  **`nohup`** *my_daemon* **`> /dev/null &`**

## Color

- Some color devices may reverse a color request. For example, `highlight_bar=red` and `highlight_bar_text=green` may be displayed as “red on green” rather than “green on red.” If this happens, set `highlight_bar=green` and `highlight_bar_text=red` to produce the proper color combination. This solution will, of course, cause the problem on devices that handle color requests as expected.

## Message Line

- When the **`checkworld`** command is executed explicitly, or when a `SIGALRM` occurs after `MAILCHECK` seconds, the message line may clear. Because the reason the message line clears may not be apparent to users, documents about your application should include an explanation of this behavior.

## Syntax

- In general, FMLI does not generate messages on syntax errors. However, some of the built-in functions, such as `fmlgrep` and `fmlcut,` and the `if-then-else` statement generate their own syntax error messages. Developers should be aware that the absence of an error message does not necessarily mean that there is not a syntax error in their code.
- When creating a new menu, form, or text frame, all quotes and backquotes must match. Quoting mismatches may cause unpredictable results; the frame may never appear, or appear incorrectly.
- Prior to FMLI Release 4.0, only the `$` notation existed for variable evaluation, and that notation exhibited the behavior now defined for `$!`. For previously written FMLI applications now being run under FMLI Release 4.0 or 4.0+, a Boolean descriptor, `use_incorrect_pre4.0_behavior`, can be set in the initialization file, which causes FMLI to ignore the `$!` notation and interpret `$` in the way defined above for `$!`. The default value (if not defined in the initialization file) for this descriptor is FALSE.

## Miscellaneous

- The FMLI interpreter does not use `EOF` to exit a program. The assumption is that applications are interactive and at some point allow the user to select an item that evaluates to the **`exit`** command. Otherwise, the FMLI application will run indefinitely. Thus, if the input to FMLI is to come from a file, the file must include the **`exit`** command.
- If you are running FMLI on a system with the shell job control feature, you can interrupt an FMLI application using the `CTRL-z` key and resume it with the **`fg`** utility.

## Known Problems

### Messages

- When a mouse is used to navigate to a new frame and the mouse is pointing within the frame title or its scroll bar, the item message or field message for the current item or field may flash on the message line.
- If the evaluation of a descriptor results in a short-term message being issued, followed by the opening of another frame, then frame or permanent messages defined in the new frame will not be displayed until another key is pressed. For example, if the `done` descriptor were defined as follows:

  ```
  done=`message “I'm doing something; please wait.”;
        ...
       `open Text.confirm
  ```

  where `Text.confirm` defines

  ```
  framemsg=“Press CONT to continue”
  ```

  then when `Text.confirm` opens, the message `I'm doing something; please wait.` will continue to be displayed. Only when another key is pressed will the frame message appear.

  To avoid this, you can use the **`indicator`** command instead of the **`message`** command in the original descriptor definition. (However, you must turn off the indicator before opening the frame.) Alternatively, in the frame that is opened, the `framemsg` descriptor could be defined as follows”

  ```
  framemsg=`message -o “Press CONT to continue”`
  ```

### Screen Labels for Function Keys

- The only way to get the commands **`prev-frm`**, **`next-frm`**, **`prevpage`**, and **`nextpage`** to work on an application-defined SLK is to make the label of the SLK (using the `name` descriptor) the same as the command name. Also, if the SLK label is set to one of these (case irrelevant), that is the command that will be executed by that application-defined SLK, no matter what the `action` descriptor is coded to.

## Forms

### Multi-page forms

- An attempt to access a form page that has no active (`inactive=true`) or shown (`show=false`) fields causes the cursor to be positioned on the first field (inactive or “not shown” field) and input to the field is allowed.
- Sometimes on multi-page forms the scroll indicators (`^` and `v`) may not be shown when and after any page after the first is displayed.

### Other Form Problems

- If a user enters data in a one-line scrollable field, then navigates away from the frame without having pressed `ENTER`, the field is reset when the frame becomes current again.
- The `show`, `value`, and `inactive` descriptors are not re-evaluated for a field when a SLK is pressed, unless the `ENTER` key has been pressed after the data are entered in the field.
- When an application-defined SLK is pressed after a value has been entered in a field, the new value of the field is not set in the field variable (`F`*n*) unless the `ENTER` key has also been pressed. This is particularly relevant for forms with only one field in them.
- If an active form field is dynamically made to be inactive, then underlining is retained on those characters of the field that already have data entered in them.
- When toggling between choices, if consecutive choices are identical, the remaining choices cannot be reached. Note that this does not occur when there are enough choices to generate a menu.

## Text Frames

- The **`regex`** built-in utility used in a `text` descriptor, with a template argument that **`cat`**'s a file containing tabs and newlines, does not preserve the tabs and newlines in the opened text frame.
- A text frame with `wrap=true` set may lose its correct wrapping if the **`update`** command is issued for it.
- If the `rows` descriptor of a text frame is reduced after the frame is opened, and the frame is later updated, the display of the frame will be corrupted.
- The header in a text frame is truncated if it is longer than the `columns` descriptor specified, even if the title of the frame causes the actual size of the frame to be big enough to hold the header.

## Commands

- If a user-defined command contains a compound backquoted expression including a `run` statement, such as

  ```
  name=my_cmd
  action=`executableA;run executableB; executable C`nop
  ```

  then, if the command is selected from the command menu, everything up to and including the `run` statement (`executableB`) occurs with the command menu as the current frame; the rest of the statement (`executableC`) occurs with the previously current frame as the current frame.
- The **`reset`** command does not work in the `done` descriptor of a form.
- When a second set of SLKs are defined, the **`togslk`** command issued from the command line works once, to switch from the first set to the second, but subsequent executions of it are ignored.
- If the `rmenu` descriptor defines consecutive choices as identical (which shouldn't be done, since it serves no purpose), toggling the choices using the CHOICES function key prevents those choices after the duplicated one from being reachable.

## Built-in Utilities

### regex

- The **`regex`** built-in utility used in a `text` descriptor, with a template argument that **`cat`**'s a file containing tabs and newlines, does not preserve the tabs and newlines in the opened text frame.

### readfile

- The **`longline`** utility cannot be used to determine the longest line of a header read with the **`readfile`** utility if the `text` descriptor also contains a **`readfile`**.
- If a menu contains a **`readfile`** in a backquoted expression on a line by itself and the file read contains a series of backquoted expressions on lines by themselves, the first one of those lines is ignored. Making the first line a blank or a comment will get around this.

## Co-processing Utilities

- The frame border of a form may not complete until input is provided from the keyboard when co-processing is used.

- Input and output strings from co-processing should not use non-alphabetic printable characters, because the FMLI special characters are not correctly transmitted.
- When the `reread` descriptor is used with co-processing and the **`vsig`** utility causes the reread to occur frequently, the FMLI process may grow out of memory space.

### if-then-else

- Omitting the terminating `fi` causes the remainder of a frame definition file to be incorrectly parsed.
- The standard output of `if-then-else` cannot be redirected using the `>` operator. However, individual parts of the statement (the `then` part and the `else` part) can be.
- A null statement following a `then` followed by an `else` can cause a syntax error.

### fmlcut

- The **`fmlcut`** built-in utility reads standard input piped to it, but does not read a file redirected using `<`.

## Descriptors

- The `init` descriptor for a frame is not evaluated first, although it should be. In forms, the `value` descriptors are evaluated first, followed by the `page`, `show`, and `inactive` descriptors for each field; in menus, the `show` descriptors for items are evaluated first. If an application uses backquoted expressions in these descriptors, this ordering must be taken into consideration; `init` cannot be relied on to be evaluated first.
- When the `reread` descriptor is used with co-processing and the **`vsig`** utility causes the reread to occur frequently, the FMLI process may grow out of memory space.
- The `permanentmsg` descriptor available in the initialization file incorrectly takes precedence over a `framemsg` descriptor in a frame file.

## Interrupt Facility

- If an interrupt is generated by the user just as a frame is being displayed, corruption may occur. Use the **`refresh`** command to redraw the screen.
- If interrupts are enabled in a text frame and the `oninterrupt` descriptor does not evaluate to a valid command, then an interrupt generated just after the frame is canceled causes the frame to be canceled, when the result

should just be a beep, the standard action when the `close` descriptor does not evaluate to a valid command.

- If interrupts are enabled and the `oninterrupt` descriptor in effect does not evaluate to a valid command, then an interrupt generated when an application-defined command is being executed causes the current frame to close.
- The action specified in the `close` descriptor is not interruptible. But the action specified in the `close` descriptor in a text frame is getting interrupted. After the interrupt, the action is incorrect.

## Miscellaneous

- Broadcast messages from root (for example, ones sent using the **`wall (1)`** command) may not be readable while an FMLI application is running, and can corrupt the screen. Users of your FMLI application should be warned that this can happen and that they can access the command line with CTRL-j and execute the **`refresh`** command to redraw the screen.
- FMLI does not recognize `EOF` on its input stream, so the only way that piped input can cause an **`fmli`** execution to properly terminate is for a correct string issuing an **`exit`** command to reach FMLI at the right time (for example, CTRL-j **`exit`**, when the operation to go to the command line is recognizable). If the **`exit`** is not executed, the FMLI session will hang and must be terminated with the **`kill(1)`** utility.
- Although FMLI Release 4.0 and 4.0+ work with the shell job control feature of UNIX System V Release 4, an FMLI session cannot be started in the background, using a command of the form

  ```
  fmli Menu.1 &
  ```

  It can be started in the foreground, interrupted with the CTRL-Z key, and then put in the background using the **`bg`** utility.
- Characters that are special to FMLI (such as `\`) may require unreasonable and seemingly arbitrary escaping backslashes to be correctly assigned to variables or used as arguments.
- If FMLI is running in a dynamically resizable window (for example, under `layers` or `xterm`), it will not recognize a new window size if the window is enlarged or shrunk; this may cause corruption.

# B
# Keyboard Support

## Named Keys and Alternative Keystroke Sequences

The following table shows each of the named keys defined by FMLI and the alternative keystroke sequence that will produce the same result. CTRL represents the control key.

| NAMED KEY | ALTERNATIVE SEQUENCE | |
|---|---|---|
| BACKSPACE | CTRL-h | |
| Form:<br>moves cursor left one position, replacing the character there with a space. | Menu:<br>same as LEFT-ARROW. | Text:<br>if text frame is editable, same as for forms. Otherwise, n/a. |
| BACKTAB | CTRL-t | |
| Form:<br>moves cursor to the previous field, whether above the current field or to the left, wrapping from the first field of the form to the last. | Menu:<br>same as LEFT-ARROW. | Text:<br>n/a |
| BEG | CTRL-b | |
| Form:<br>moves cursor to the first field of the current page. | Menu:<br>moves cursor to the first item, whether currently visible or not. | Text:<br>displays first frame full of text. |
| CLEAR<br>CLEAR-LINE | CTRL-y<br>CTRL-y | |
| Form:<br>clears the current line. | Menu:<br>n/a | Text:<br>if text frame is editable, same as for forms. Otherwise, n/a. |
| CLEAR-EOL | CTRL-f y | |
| Form:<br>clears the current line from the current cursor position to the end of the line. | Menu:<br>n/a | Text:<br>if text frame is editable, same as for forms. Otherwise, n/a. |

| NAMED KEY ALTERNATIVE SEQUENCE | | |
|---|---|---|
| COMMAND LINE CTRL-j or CTRL-f c | | |
| Form:<br>moves cursor to command line. | Menu:<br>same. | Text:<br>same. |
| DEL CTRL-x<br>DELETE-CHAR CTRL-x | | |
| Form:<br>deletes the character under cursor and closes the gap. | Menu:<br>n/a | Text:<br>if text frame is editable, same as for forms. Otherwise, n/a. |
| DELETE-LINE CTRL-k | | |
| Form:<br>in multi-line fields, deletes the current line and closes the gap. In single-line fields, same as CLEAR-LINE. | Menu:<br>n/a | Text:<br>if text frame is editable, same as for forms. Otherwise, n/a. |
| DOWN-ARROW CTRL-d | | |
| Form:<br>in a single-line field, moves cursor to the next field below the current one, wrapping from the last field of the column to the first. In a multi-line field, it moves cursor to the next line; on the last line, it moves cursor to the next field below the current one. | Menu:<br>moves cursor down one item, wrapping to the top of the column in a single-column menu, the top of the next column in a multi-column menu. On the last item in the last column of a multi-column menu, it wraps to the top of the first column. | Text:<br>moves cursor down one line. It does not wrap. |
| END CTRL-e | | |
| Form:<br>moves cursor to the last field of the current page. | Menu:<br>moves cursor to the last item, whether currently visible or not. | Text:<br>displays last frame full of text. |
| ENTER CTRL-m | | |
| Form:<br>in a single-line field, moves cursor to the next field, whether below the current field or to the right, wrapping from the last field of the form to the first. In a multi-line field, it moves the cursor to the next line. This key cannot be used to navigate from a multi-line field (it scrolls on the last line if the field is scrollable, stops and beeps if it is non-scrollable). Validation occurs even if no data have been entered or modified in the field since it became current. | Menu:<br>selects the current item in a single-select menu, the marked items in a multi-select menu. | Text:<br>moves cursor down one line. It does not wrap. |

| NAMED KEY | ALTERNATIVE SEQUENCE | | | |
|---|---|---|---|---|
| HOME | CTRL-f b | Form:<br>moves cursor to the first character of the current field. | Menu:<br>moves cursor to the first item currently visible. | Text:<br>displays first frame full of text. |
| HOME-DOWN | CTRL-f e | Form:<br>moves cursor to the last character of the current field. | Menu:<br>moves cursor to the last item currently visible. | Text:<br>displays last frame full of text. |
| INSERT-CHAR | CTRL-a | Form:<br>inserts a space to the left of the character under cursor and moves cursor over the space. The next character entered will replace the space. | Menu:<br>n/a | Text:<br>if text frame is editable, same as for forms. Otherwise, n/a. |
| INSERT-LINE | CTRL-o | Form:<br>in a multi-line field, if space is available, opens a line below the current line and puts cursor on that line. Otherwise, n/a. | Menu:<br>n/a | Text:<br>if text frame is editable, same as for forms. Otherwise, n/a. |
| LEFT-ARROW | CTRL-l | Form:<br>moves cursor non-destructively one character to the left. It does not wrap to the previous field, or the previous line in a multi-line field. | Menu:<br>moves cursor left one item in a multi-column menu, up one item in a single-column menu. In a multi-column menu, it does not wrap. In a single-column menu, it wraps to the bottom of the column. | Text:<br>same as for forms. |
| MARK | CTRL-f m | Form:<br>n/a | Menu:<br>In a multi-select menu, marks the item to be selected. In a single-select menu, n/a. | Text:<br>n/a |
| NEXT | CTRL-n | Form:<br>same as TAB. | Menu:<br>same as DOWN-ARROW. | Text:<br>n/a |
| PAGE-DOWN | CTRL-w | Form:<br>in a multi-page form, moves cursor to the first field of the next page. | Menu:<br>in a scrollable menu, moves cursor to the first item of the next frame full of items and displays that frame full, unless there are fewer than 10 items, in which case the terminal rings (or flashes). | Text:<br>in a scrollable text frame, moves cursor to the first line of the next frame full of text and displays that frame full, preserving two lines from the current frame. |

| NAMED KEY | ALTERNATIVE SEQUENCE | |
|---|---|---|
| PAGE-UP CTRL-v | | |
| Form:<br>in a multi-page form, moves cursor to the first field of the previous page. | Menu:<br>in a scrollable menu, moves cursor to the first item of the previous frame full of items and displays that frame full, unless there are fewer than 10 items, in which case the terminal rings (or flashes). | Text:<br>in a scrollable text frame, moves cursor to the first line of the previous frame full of text and displays that frame full, preserving two lines from the current frame. |
| PREV CTRL-p | | |
| Form:<br>same as BACKTAB. | Menu:<br>same as UP-ARROW. | Text:<br>n/a |
| RESET CTRL-f r | | |
| Form:<br>resets a field to its default value. | Menu:<br>n/a | Text:<br>n/a |
| RETURN CTRL-m | | |
| Form:<br>same as ENTER. | Menu:<br>same as ENTER. | Text:<br>same as ENTER. |
| RIGHT-ARROW CTRL-r | | |
| Form:<br>moves cursor non-destructively one character to the right. It does not wrap to the next field, or the next line in a multi-line field. | Menu:<br>moves cursor right one item in a multi-column menu, down one item in a single-column menu. In a multi-column menu, it does not wrap. In a single-column menu, it wraps to the top of the column. | Text:<br>same as for forms. |
| SCREEN-LABELED KEYS<br>CTRL-f 1 … CTRL-f 8 | | |
| Form:<br>performs the action assigned to the function key by default or by programmer. | Menu:<br>same. | Text:<br>same. |
| SCROLL-DOWN CTRL-f d | | |
| Form:<br>rolls the contents of a multi-line scrollable field down by the number of lines displayed. | Menu:<br>rolls the contents of a scrollable menu down one line, without moving the cursor. | Text:<br>rolls the contents of a scrollable text frame down one line, without moving the cursor. |
| SCROLL-UP CTRL-f u | | |
| Form:<br>rolls the contents of a multi-line scrollable field up by the number of lines displayed. | Menu:<br>rolls the contents of a scrollable menu up one line, without moving the cursor. | Text:<br>rolls the contents of a scrollable text frame up one line, without moving the cursor. |

| NAMED KEY | ALTERNATIVE SEQUENCE | |
|---|---|---|
| SPACEBAR | none | |
| Form: replaces the current character with a space and moves cursor one character to the right. | Menu: same as RIGHT-ARROW | Text: if text frame is editable, same as for forms. Otherwise, n/a. |
| TAB | `CTRL-i` | |
| Form: moves cursor to the next field, whether below the current field or to the right, wrapping from the last field of the form to the first. | Menu: same as RIGHT-ARROW. | Text: n/a |
| UP-ARROW | `CTRL-u` | |
| Form: in a single-line field, moves cursor to the previous field above the current one, wrapping from the first field of the column to the last. In a multi-line field, it moves cursor to the previous line; on the first line, it moves cursor to the previous field above the current one. | Menu: moves cursor up one item, wrapping to the bottom of the column in a single-column menu, the bottom of the previous column in a multi-column menu. On the first item in the first column of a multi-column menu, it wraps to the bottom of the last column. | Text: moves cursor up one line. It does not wrap. |

## Automatic Function Key Downloading

FMLI applications rely heavily on the use of function keys `F1` through `F8`. Because these keys are not available on some terminals or are not assigned default escape sequences that `curses` can use, FMLI provides alternative keystroke sequences, `CTRL-f 1` through `CTRL-f 8`, respectively, whose use is equivalent to that of function keys. Some terminals, such as the AT&T 5620 and 630 terminals, as defined in their **`terminfo(4)`** entries, do not assign default escape sequences to these function keys, but can download strings into them. FMLI will automatically download the alternative keystroke sequences if either of the following is true:

- the environment variable `LOADPFK` is set to `yes`, `true`, or the null string;
- the user chooses FMLI function key downloading at a prompt given during initialization of the application. This prompt is not given if `LOADPFK` is set to any value.

In either case, the alternative sequences replace any previously defined strings for the function keys. You can, however, restore the previously defined strings if you have taken the precaution of storing them as an executable shell script. Such a script might use the **`tput`** utility as follows:

```
tput pfx 1 'string-for-function-key-1'
tput pfx 2 'string-for-function-key-2'
tput pfx 3 'string-for-function-key-3'
tput pfx 4 'string-for-function-key-4'
tput pfx 5 'string-for-function-key-5'
tput pfx 6 'string-for-function-key-6'
tput pfx 7 'string-for-function-key-7'
tput pfx 8 'string-for-function-key-8'
```

If you execute this script after exiting from an FMLI application, the stored function key definitions will be downloaded.

An FMLI application will execute this script automatically if it is named **`.restorePFKs`**. FMLI looks for a file named **`.restorePFKs`** first in the current directory, then in `$HOME`. If it finds such a file, the FMLI application displays the message

```
Running the shell script in filename
to restore function key settings
```

where *filename* is either **`.restorePFKs`** or **`$HOME/.restorePFKs`**.

**NOTE**

The **`tput`** utility must be UNIX System V Release 4 or later.

Referencing a **`terminfo`** entry for the AT&T 5620 terminal from a **`terminfo`** database older than UNIX System V Release 4 will cause the alternative keystroke sequences to be incorrectly downloaded. Not all the alternative sequences will be downloaded, and garbage may be output to the screen. This can occur if a user maintains a private variant of the **`terminfo`** entry. Such a user should not choose function key downloading.

# C
# TAM Transition Library

## Introduction

Character mode applications that run under the Terminal Access Method (TAM) on the UNIX PC can now run under ETI with a wide range of terminals. This appendix explains how to use the TAM transition library, the source of this portability. In addition, it explains how you can eventually rewrite your TAM application programs to run more efficiently under ETI without the TAM transition library.

## Compiling and Running TAM Applications under ETI

The TAM transition library consists of a header file **`tam.h`** and a set of library routines. The file **`tam.h`** translates between TAM routines and equivalent sets of low-level ETI routines. For example, the TAM function `wcreate` is mapped to the conversion library function `TAMwcreate`, which consists of a series of low-level ETI calls, such as `newwin` and `subwin`.

To use the TAM transition library, be sure to include the standard TAM header file **`tam.h`** in your application program. So at the beginning of your TAM application program, you should already have

```
#include <tam.h>  /* as usual, for TAM calls */
```

Next, you recompile and link your application program, say **`tamprog.c`**, to form an executable, as follows:

```
cc -I /usr/add-on/include tamprog.c -ltam -lcurses \
   -o executable_name
```

Note the use of the **`-I`** option, which tells the compiler where to find the TAM header files. The two uses of the **`-l`** option link the requisite library subroutines, the TAM transition library and the low-level ETI library.

Alternatively, you might separately compile one or more TAM application files (say, **`tam1.c`**, **`tam2.c`**, and **`main.c`**) and later link them to form an executable program.

```
cc -c -I /usr/add-on/include/ tam1.c
            /* compile files individually */
cc -c -I /usr/add-on/include/ tam2.c
cc -c -I /usr/add-on/include/ main.c
            /* link objects to form executable */
```

```
cc -o executable_name tam1.o tam2.o main.o -ltam -lcurses
```

Note that the **-I** option is required for the compilation of any file that uses the TAM library.

## Tips for Polishing TAM Application Programs Running under ETI

To enable the code in your TAM application program to run smoothly under ETI, you should do the following:

- remove code that would be executed if a low-level `iswind` function call returned a non-zero value, that is, TRUE. Under the TAM transition library, `iswind` always returns FALSE.
- remove all TAM calls to mouse management routines and the calls `wicon`, `wicoff`, and `wrastop`, because they will translate to null operations.
- remove all machine-specific code, because the TAM transition library does not translate system calls specifically tailored to the UNIX PC or calls (such as **`ioctl(2)`**) that have no meaning under ETI. These calls fail under the TAM transition library on all machines except the UNIX PC.
- note that all calls to **`track(3T)`** map to the low-level function `wgetc`.
- remove all references to TAM calls that bear the same name as ETI calls because calls that have the same names in both systems have different effects.
- remove all arbitrary ANSI escape sequences for display output. For example, the TAM transition library does not recognize the escape sequence used on the UNIX PC in the command `echo “\ 33[J”`, which clears the screen. Instead, you should use equivalent ETI routines (here, `clear`).

Eliminating the superfluous code in the first three cases reduces your program's size and execution time.

## How the TAM Transition Library Works

The TAM Transition Library translates between TAM function calls and low-level ETI function calls. It also ensures that escape and control sequences entered at a terminal's keyboard are properly interpreted.

## Translations from TAM Calls to ETI Calls

The table in Table C-1 summarizes the translation of TAM to low-level ETI (**`curses`**) functions. Eventually, if you want to rewrite your TAM applications to make ETI calls directly and to run more efficiently, you can use this table as a guide.

**Table C-1. Translations from TAM to ETI Function Calls**

| TAM Function | Low-level ETI (**`curses(3curses)`**) Equivalent |
|---|---|
| winit | Call `initscr`. |
| wexit | Call `endwin` and `exit`. |
| iswind | Return FALSE. |
| wcreate | Call `newwin` or `new_panel`. |
| wdelete | Call `delwin` or `del_panel`. |
| wselect | Call `touchwin` and `wrefresh,` then update the list of windows to indicate the new ordering. |
| wgetsel | Call `top_panel` or `bottom_panel` with NULL pointer. |
| wgetstat | Call `getyx`, `getmaxyx`, or `getbegyx`. |
| wsetstat | Call `del_panel`, then `new_panel`. |
| wputc | Call `waddch`. |
| wputs | Call `waddstr`. |
| wprintf | Call `wprintw`. |
| wslk | Create small window at bottom and use `curses` routines with `wprintw`. |
| wcmd | The character string passed by `wcmd` is copied to the bottom of the screen. |
| wprompt | The character string passed by `wprompt` is copied to the bottom of the screen. |
| wlabel | The character string is printed in the upper left corner of the specified window. |
| wrefresh | Call `wrefresh`. If the window index is -1, all windows should be refreshed in the appropriate order. |
| wuser | This functionality is not necessary. Remove this from your code. |
| wgoto | Call `wmove`. |
| wgetpos | Call `getyx`. |
| wgetc | Call `wgetch`. Character translation from ETI to ANSI may be required, depending on the current `keypad` mode. |
| kcodemap | This functionality is not necessary. Remove this from your code. |
| keypad | Call `keypad`. |

**Table C-1. Translations from TAM to ETI Function Calls (Cont.)**

| TAM Function | Low-level ETI (**`curses(3curses)`**) Equivalent |
|---|---|
| wsetmouse | This is a null operation. |
| wgetmouse | This is a null operation. |
| wreadmouse | This is a null operation. |
| wprexec | Call `erase` and `refresh`. |
| wpostwait | Call `wrefresh` for each window in the window list. |
| wnl | The functionality of this routine is not supported by `curses`. |
| wicon | This is a null operation. |
| wicoff | This is a null operation. |
| wrastop | This is a null operation. |
| track | Call `wgetch`. |
| initscr | Call `initscr`. |
| nl | The functionality of this routine is not supported by `curses`. |
| nonl | The functionality of this routine is not supported by `curses`. |
| cbreak | Call `cbreak`. |
| nocbreak | Call `nocbreak`. |
| echo | Call `echo`. |
| noecho | Call `noecho`. |
| insch | Call `insch`. |
| getch | Call `getch`. |
| flushinp | Call `flushinp`. |
| attron | Call `attron`. |
| attroff | Call `attroff`. |
| savetty | Call `savetty`. |
| resetty | Call `resetty`. |
| addch | Call `addch`. |
| addstr | Call `addstr`. |
| beep | Call `beep`. |
| clear | Call `clear`. |
| clearok | This is a null operation. |
| clrtobot | Call `clrtobot`. |
| clrtoeol | Call `clrtoeol`. |
| delch | Call `delch`. |
| deleteln | Call `deleteln`. |

**Table C-1. Translations from TAM to ETI Function Calls (Cont.)**

| TAM Function | Low-level ETI (`curses(3curses)`) Equivalent |
|---|---|
| erase | Call `erase`. |
| flash | Call `flash`. |
| getyx | Call `wgetyx`. |
| insertln | Call `insertln`. |
| leaveok | This is a null operation. |
| move | Call `move`. |
| mvaddch | Call `move` and `addch`. |
| mvaddstr | Call `move` and `addstr`. |
| mvinch | Call `move` and `inch`. |
| nodelay | Call `nodelay`. |
| wndelay | Call `nodelay`. |
| refresh | Call `refresh`. |
| resetterm | Call `resetterm`. |
| baudrate | Call `baudrate`. |
| endwin | Call `endwin`. |
| fixterm | Call `fixterm`. |
| printw | Call `printw`. |

Because the high-level TAM functions in the table in Table C-2 make calls only to the low-level functions in the previous table, you can continue to use those high-level TAM functions in your application programs as well. However, with ETI, you cannot use other TAM high-level functions such as `wtargeton`.

**Table C-2. TAM High-level Functions**

| Usable TAM High-level Functions | | |
|---|---|---|
| form | menu | message |
| pb_empty | pb_gets | adf_gttok |
| pb_open | pb_check | pb_seek |
| pb_name | pb_puts | pb_weof |
| pb_gbuf | adf_gtwrd | adf_gtxcd |
| wind | exhelp | |

## The TAM Transition Keyboard Subsystem

Both TAM and ETI use a set of virtual function keys that translate between an escape character sequence entered at the keyboard and a bit pattern inside the machine. Under the TAM transition library, the TAM virtual key values are translated into ETI virtual key values.

The table in Table C-3 lists these equivalent virtual key values. Entering the escape sequence listed in the left column will generate the corresponding TAM virtual function key value given in the middle column. The right column lists the ETI equivalent of the TAM virtual key and is for reference only.

**Table C-3. Translation Between TAM Escape Sequences and Virtual Key Values**

| TAM Escape Sequence | Virtual Key Value | |
|---|---|---|
| | TAM | ETI |
| ESC-! | s_F1 | KEY_F(8) |
| ESC-@ | s_F2 | KEY_F(9) |
| ESC-# | s_F3 | KEY_F(10) |
| ESC-$ | s_F4 | KEY_F(11) |
| ESC-% | s_F5 | KEY_F(12) |
| ESC-^ | s_F6 | KEY_F(13) |
| ESC-& | s_F7 | KEY_F(14) |
| ESC-* | s_F8 | KEY_F(15) |
| ESC-f1 | PF1 | KEY_F(16) |
| ESC-f2 | PF2 | KEY_F(17) |
| ESC-f4 | PF3 | KEY_F(18) |
| ESC-f4 | PF4 | KEY_F(19) |
| ESC-f5 | PF5 | KEY_F(20) |
| ESC-f6 | PF6 | KEY_F(21) |
| ESC-f7 | PF7 | KEY_F(22) |
| ESC-f8 | PF8 | KEY_F(23) |
| ESC-f9 | PF9 | KEY_F(24) |
| ESC-f0 | PF10 | KEY_F(25) |
| ESC-f- | PF11 | KEY_F(26) |
| ESC-f= | PF12 | KEY_F(27) |
| ESC-1 | F1 | KEY_F(0) |
| ESC-2 | F2 | KEY_F(1) |

**Table C-3. Translation Between TAM Escape Sequences and Virtual Key Values (Cont.)**

| TAM Escape Sequence | Virtual Key Value | |
|---|---|---|
| | TAM | ETI |
| ESC-3 | F3 | KEY_F(2) |
| ESC-4 | F4 | KEY_F(3) |
| ESC-5 | F5 | KEY_F(4) |
| ESC-6 | F6 | KEY_F(5) |
| ESC-7 | F7 | KEY_F(6) |
| ESC-8 | F8 | KEY_F(7) |
| ESC-bg | Beg | KEY_BEG |
| ESC-BG | s_Beg | KEY_SBEG |
| ESC-br | Break | KEY_BREAK |
| ESC-bw | Back | KEY_LEFT |
| ESC-BW | s_Back | KEY_SLEFT |
| ESC-ce | Clear | KEY_CLEAR |
| ESC-CE | Clear | KEY_CLEAR |
| ESC-ci | ClearLine | KEY_EOL |
| ESC-CI | s_ClearLine | KEY_SEOL |
| ESC-cl | Close | KEY_CLOSE |
| ESC-CL | Close | KEY_CLOSE |
| ESC-cm | Cmd | KEY_COMMAND |
| ESC-CM | s_Cmd | KEY_SCOMMAND |
| ESC-cn | Cancl | KEY_CANCEL |
| ESC-CN | s_Cancl | KEY_SCANCEL |
| ESC-cp | Copy | KEY_COPY |
| ESC-CP | s_Copy | KEY_SCOPY |
| ESC-cr | Creat | KEY_CREATE |
| ESC-CR | s_Creat | KEY_SCREATE |
| ESC-dc | DleteChar | KEY_DC |
| ESC-Del | DleteChar | KEY_DC |
| ESC-DC | s_DleteChar | KEY_SDC |
| ESC-dl | Dlete | KEY_DL |
| ESC-DL | s_Dlete | KEY_SDL |
| ESC-dn | Down | KEY_DOWN |

**Table C-3. Translation Between TAM Escape Sequences and Virtual Key Values (Cont.)**

| TAM Escape Sequence | Virtual Key Value | |
|---|---|---|
| | TAM | ETI |
| ESC-DN | RollDn | KEY_SF |
| ESC-en | End | KEY_END |
| ESC-EN | s_End | KEY_SEND |
| ESC-ESC | Esc | none |
| ESC-ex | Exit | KEY_EXIT |
| ESC-EX | s_Exit | KEY_SEXIT |
| ESC-fi | Find | KEY_FIND |
| ESC-FI | s_Find | KEY_SFIND |
| ESC-fw | Forward | KEY_RIGHT |
| ESC-FW | s_Forward | KEY_SRIGHT |
| ESC-hl | Help | KEY_HELP |
| ESC-? | Help | KEY_HELP |
| ESC-HL | s_Help | KEY_SHELP |
| ESC-hm | Home | KEY_HOME |
| ESC-HM | s_Home | KEY_SHOME |
| ESC-im | InputMode | KEY_IC |
| ESC-NJ | s_InputMode | KEY_SIC |
| ESC-mk | Mark | KEY_MARK |
| ESC-MK | Slect | KEY_SELECT |
| ESC-ms | Msg | KEY_MESSAGE |
| ESC-MS | s_Msg | KEY_SMESSAGE |
| ESC-mv | Move | KEY_MOVE |
| ESC-MV | s_Move | KEY_SMOVE |
| ESC-nx | Next | KEY_NEXT |
| ESC-NX | s_Next | KEY_SNEXT |
| ESC-op | Open | KEY_OPEN |
| ESC-OP | Close | KEY_CLOSE |
| ESC-ot | Opts | KEY_OPTIONS |
| ESC-OT | s_Opts | KEY_SOPTIONS |
| ESC-pg | Page | KEY_NPAGE |
| ESC-PG | s_Page | KEY_PPAGE |

**Table C-3. Translation Between TAM Escape Sequences and Virtual Key Values (Cont.)**

| TAM Escape Sequence | Virtual Key Value | |
|---|---|---|
| | TAM | ETI |
| ESC-pr | Print | KEY_PRINT |
| ESC-PR | s_Print | KEY_SPRINT |
| ESC-pv | Prev | KEY_PREVIOUS |
| ESC-PV | s_Prev | KEY_SPREVIOUS |
| ESC-rd | RollDn | KEY_SF |
| ESC-RD | RollDn | KEY_SF |
| ESC-re | Ref | KEY_REFERENCE |
| ESC-RE | Rstrt | KEY_RESTART |
| ESC-rf | Rfrsh | KEY_REFRESH |
| ESC-RF | Clear | KEY_CLEAR |
| ESC-rm | Rsume | KEY_RESUME |
| ESC-RM | s_Rsume | KEY_SRSUME |
| ESC-ro | Redo | KEY_REDO |
| ESC-RO | s_Redo | KEY_SREDO |
| ESC-rp | Rplac | KEY_REPLACE |
| ESC-RP | s_Rplac | KEY_SREPLACE |
| ESC-rs | Rstrt | KEY_REFERENCE |
| ESC-RS | Rstrt | KEY_RESTART |
| ESC-ru | RollUp | KEY-SR |
| ESC-RU | RollUp | KEY_SR |
| ESC-sl | Slect | KEY_SELECT |
| ESC-SL | Slect | KEY_SELECT |
| ESC-ss | Suspd | KEY_SUSPEND |
| ESC-SS | s_Suspd | KEY_SSUSPEND |
| ESC-sv | Save | KEY_SAVE |
| ESC-SV | s_Save | KEY_SSAVE |
| ESC-ud | Undo | KEY_UNDO |
| ESC-UD | s_Undo | KEY_SUNDO |
| ESC-up | Up | KEY_UP |
| ESC-UP | RollUp | KEY_SR |

Some keyboards have one or more keys that emit escape sequences that are identical to the UNIX PC keyboard sequences but do not match in terms of functionality. The function of an operationally incompatible key will always map to its **`terminfo`** specification. The TAM specific function implied by the same escape sequence will be accessible through the technique describe above. Mechanisms in **`curses(3curses)`** automatically handle timing conflicts between actual keyboard function keys and UNIX PC keyboard escape sequences.

# D
# ETI Program Examples

## Program Examples

The following programs demonstrate uses of low-level ETI (**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 that are all discussed in this document.

Second, it also uses some **curses** routines that are not discussed in this document. For example, the function to write out a file uses the `mvinch` routine, which returns a character in a window at a given position. The data structure used to write out a file does not keep track of the number of characters in a line or the number of lines in the file, so trailing blanks are eliminated when the file is written. The program also uses the `insch`, `delch`, `insertln`, and `deleteln` routines. These functions insert and delete a character or line. See the **curses(3curses)** manual pages 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 (that is, escape sequences) to control the terminal and have special keys that send escape sequences to the computer. If a computer receives an escape from a terminal, it cannot tell whether the user depressed the escape key or whether a special key was pressed.

**`editor`** and other **`curses`** programs handle the ambiguity by setting a timer. If another character is received during this time, and if that character might be the beginning of a special key, the program reads more input until either a full special key is read, the time out is reached, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press escape, then to type another key quickly, which causes the **`curses`** program to think a special key has been pressed. Also, a pause occurs until the escape can be passed to the user program, resulting in a slower response to the escape key.

Many existing programs use escape as a fundamental command, which cannot be changed without infuriating a large class of users. These programs cannot make use of special keys without dealing with this ambiguity, and at best must resort to a time-out solution. The moral is clear: when designing your **`curses`** programs, avoid the escape key.

```
 /* editor: A screen-oriented editor.  The user
  * interface is similar to a subset of vi.
  * The buffer is kept in stdscr to simplify
  * the program.
  */

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

#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;

    standout();
    mvaddstr(LINES - 1, COLS - 20, "INPUT MODE");
    standend();
    move(row, col);
    refresh();
    for (;;)
    {
        c = getch();
        if (c == CTRL('D') || c == KEY_EIC)
            break;
        insch(c);
        move(row, ++col);
        refresh();
    }
    move(LINES - 1, COLS - 20);
    clrtoeol();
    move(row, col);
    refresh();
}
```

## The highlight Program

This program illustrates a use of the routine `attrset`. `highlight` reads a text file and uses embedded escape sequences to control attributes. `\U` turns on underlining, `\B` turns on bold, and `\N` restores the default output attributes.

Note the first call to `scrollok`, a routine that we have not previously discussed (see the **`curses(3curses)`** manual pages). This routine allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw past the bottom of the screen, `scrollok` automatically scrolls the terminal up a line and calls `refresh`.

```
/*
 * highlight: a program to turn \U, \B, and
 * \N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */

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

main(argc, argv)
int argc;
char **argv;
{
    FILE *fd;
    int c, c2;
    void exit(), perror();

    if (argc != 2)
    {
        fprintf(stderr, “Usage: highlight file\n”);
        exit(1);
    }

    fd = fopen(argv[1], “r”);

    if (fd == NULL)
    {
        perror(argv[1]);
        exit(2);
    }

    initscr();
    scrollok(stdscr, TRUE);
    nonl();
    while ((c = getc(fd)) != EOF)
    {
        if (c == '\\')
        {
            c2 = getc(fd);
            switch (c2)
            {
            case 'B':
                attrset(A_BOLD);
                continue;
            case 'U':
                attrset(A_UNDERLINE);
                continue;
            case 'N':
                attrset(0);
                continue;
            }

            addch(c);
            addch(c2);
        }
        else
            addch(c);
    }
    fclose(fd);
    refresh();
    endwin();
    exit(0);
}
```

## The scatter Program

This program takes the first `LINES - 1` lines of characters from the standard input and displays the characters on a terminal screen in a random order. For this program to work properly, the input file should not contain tabs or non-printing characters.

```
/*
 *  The scatter program.
 */

#include<curses.h>
#include<sys/types.h>

extern time_t time();

#define MAXLINES 120
#define MAXCOLS  160
char s[MAXLINES][MAXCOLS];/* Screen Array */
int  T[MAXLINES][MAXCOLS];/* Tag Array - Keeps track of   *
                           * the number of characters     *
                           * printed and their positions. */

main()
{
    register int row = 0,col = 0;
    register int c;
    int char_count = 0;
    time_t t;
    void exit(), srand();

    initscr();
    for(row = 0;row < MAXLINES;row++)
        for(col = 0;col < MAXCOLS;col++)
            s[row][col]=' ';

    col = row = 0;
    /* Read screen in */
    while ((c=getchar()) != EOF && row < LINES ) {

        if(c != '\n')
        {
            /* Place char in screen array */
            s[row][col++] = c;
            if(c != ' ')
                char_count++;
        }
        else
        {
            col = 0;
            row++;
        }
    }

    time(&t);/* Seed the random number generator */
    srand((unsigned)t);

    while (char_count)
    {
        row = rand() % LINES;
        col = (rand() >> 2) % COLS;
        if (T[row][col] != 1 && s[row][col] != ' ')
        {
            move(row, col);
            addch(s[row][col]);
            T[row][col] = 1;
            char_count--;
            refresh();
        }
    }
    endwin();
    exit(0);
}
```

## The show Program

This program pages through a file, showing one screen of its contents each time you depress the space bar. The program calls `cbreak` so that you can depress the space bar without having to hit return; it calls `noecho` to prevent the space from echoing on the screen. The `nonl` routine, which we have not previously discussed, is called to enable more cursor optimization. The `idlok` routine, which we also have not discussed, is called to allow insert and delete line. (See the **`curses(3curses)`** pages for more information about these routines). Also notice that `clrtoeol` and `clrtobot` are called.

By creating an input file for `show` made up of screen-sized (about 24 lines) pages, each varying slightly from the previous page, nearly any exercise for a **`curses`** program can be created. This type of input file is called a show script.

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

main(argc, argv)
int argc;
char *argv[];
{
    FILE *fd;
    char linebuf[BUFSIZ];
    int line;
    void done(), perror(), exit();

    if (argc != 2)
    {
        fprintf(stderr, “usage: %s file\n”, argv[0]);
        exit(1);
    }

    if ((fd=fopen(argv[1], “r”)) == NULL)
    {

        perror(argv[1]);
        exit(2);
    }
    signal(SIGINT, done);

    initscr();
    noecho();
    cbreak();
    nonl();
    idlok(stdscr, TRUE);

    while(1)
    {
        move(0,0);
        for (line = 0; line < LINES; line++)
        {
            if (!fgets(linebuf, sizeof linebuf, fd))
            {
                clrtobot();
                done();
            }
            move(line, 0);
            printw(“%s”, linebuf);
        }
        refresh();
        if (getch() == 'q')
            done();
    }
}

void done()
{
    move(LINES - 1, 0);
    clrtoeol();
    refresh();
    endwin();
    exit(0);
}
```

## The two Program

This program pages through a file, writing one page to the terminal from which the program is invoked and the next page to the terminal named on the command line. It then waits for a space to be typed on either terminal and writes the next page to the terminal at which the space is typed.

The **`two`** program is just a simple example of a two-terminal **`curses`** program. It does not handle notification; instead, it requires the name and type of the second terminal on the command line. As written, the command “`sleep 100000`” must be typed at the second terminal to put it to sleep while the program runs, and the user of the first terminal must have both read and write permission on the second terminal.

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

SCREEN *me, *you;
SCREEN *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
int argc;
char **argv;
{
    void done(), exit();
    unsigned sleep();
    char *getenv();
    int c;

    if (argc != 4)
    {
         fprintf(stderr, “Usage: two othertty otherttytype inputfile\n”);
         exit(1);
    }
    fd = fopen(argv[3], “r”);
    fdyou = fopen(argv[1], “w+”);
    signal(SIGINT, done);/* die gracefully */

    me = newterm(getenv(“TERM”), stdout, stdin);  /* initialize my tty */
    you = newterm(argv[2], fdyou, fdyou);/* Initialize the other terminal */

    set_term(me);/* Set modes for my terminal */
    noecho();/* turn off tty echo */
    cbreak();/* enter cbreak mode */
    nonl();  /* Allow linefeed */
    nodelay(stdscr, TRUE);/* No hang on input */

    set_term(you);/* Set modes for other terminal */
    noecho();
    cbreak();
    nonl();
    nodelay(stdscr,TRUE);

    /* Dump first screen full on my terminal */
    dump_page(me);

    /* Dump second screen full on the other terminal */
    dump_page(you);

    for (;;)/* for each screen full */
    {
         set_term(me);
         c = getch();
         if (c == 'q')/* wait for user to read it */
            done();
         if (c == ' ')
            dump_page(me);

         set_term(you);
         c = getch();
         if (c == 'q')/* wait for user to read it */
            done();
         if (c == ' ')
            dump_page(you);
         sleep(1);
    }
}

dump_page(term)
  SCREEN *term;
{
    int line;

    set_term(term);
    move(0, 0);
    for (line = 0; line < LINES - 1; line++) {
         if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
            clrtobot();
            done();
         }
         mvaddstr(line, 0, linebuf);
    }
    standout();
    mvprintw(LINES - 1, 0, “--More--”);
    standend();
    refresh();/* sync screen */
}
/*
   Clean up and exit.
 */
void done()
{
    /* Clean up first terminal */
    set_term(you);
    move(LINES - 1,0);/* to lower left corner */

    clrtoeol();/* clear bottom line */
    refresh();/* flush out everything */
    endwin();/* curses cleanup */
    delscreen(); /* remove screen */

    /* 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 */
    delscreen();/* remove screen */
    exit(0);
}
```

## The window Program

This example program demonstrates the use of multiple windows. The main display is kept in `stdscr`. When you want to put something other than what is in `stdscr` on the physical terminal screen temporarily, a new window is created covering part of the screen. A call to `wrefresh` for that window causes it to be written over the `stdscr` image on the terminal screen. Calling `refresh` on `stdscr` results in the original window being redrawn on the screen. Note the calls to the `touchwin` routine (which we have not discussed — see the **`curses(3curses)`** manual pages) that occur before writing out a window over an existing window on the terminal screen. This routine prevents screen optimization in a **`curses`** program. If you have trouble refreshing a new window that overlaps an old window, it may be necessary to call `touchwin` for the new window to get it completely written out.

```
#include <curses.h>

WINDOW *cmdwin;

main()

{

    int i, c;
    char buf[120];
    void exit();

    initscr();
    nonl();
    noecho();
    cbreak();

    cmdwin = newwin(3, COLS, 0, 0);/* top 3 lines */
    for (i = 0; i < LINES; i++)
        mvprintw(i, 0, “This is line %d of stdscr”, i);

    for (;;)

    {
        refresh();
        c = getch();
        switch (c)

        {

        case 'c':/* Enter command from keyboard */
            werase(cmdwin);
            wprintw(cmdwin, “Enter command:”);
            wmove(cmdwin, 2, 0);
            for (i = 0; i < COLS; i++)
                waddch(cmdwin, '-');
            wmove(cmdwin, 1, 0);
            touchwin(cmdwin);
            wrefresh(cmdwin);
            wgetstr(cmdwin, buf);
            touchwin(stdscr);

            /*
             * The command is now in buf.
             * It should be processed here.
             */

        case 'q':
            endwin();
            exit(0);
        }

    }

}
```

## The colors Program

This program creates two windows. All characters displayed in the first window will be in red, on a blue background. All characters displayed in the second window will be in yellow, on a magenta background.

```
#include <curses.h>

#define  PAIR1    1
#define  PAIR2    2

main()
{
    WINDOW *win1, *win2;

    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”);
        wnoutrefresh (win1);
        wnoutrefresh (win2);
        doupdate();

        /* wait for any key before terminating */

        wgetch (win2);
    }

    endwin();
}
```

# Index

**Spine for 1.5” Binder**

**Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold**

**Volume Number (if any): Helvetica, 24 pt, Bold**

**Volume Name (if any): Helvetica, 18 pt, Bold**

**Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title**

**Bar: 1" x 1/8" beginning 1/4" in from either side**

**Part Number: Helvetica, 6 pt, centered, 1/8" up**

**PowerMAX OS**

**Programmer**

**Character User’s Interface Programming**

**0890424**