AT&T

UNIX® System V/386 Release 3.2

FMLI Programmer's Guide

Table of Contents

i Introduction

Introduction

What this Document Covers

The purpose of this document is to explain:

- The capabilities of the Form and Menu Language Interpreter (FMLI)
- The syntax of the Form and Menu Language
- How the Interpreter interfaces with the UNIX system.

Prerequisite Knowledge

Before attempting to use FMLI, you should be familiar with the following:

- UNIX System V Operating System
- Shell scripts and programming
- UNIX system documentation conventions
- The *Release Notes.*

How to Use this Document

This document is written for the application developer who already knows about the UNIX system and shell programming. Thus, its purpose is to familiarize the developer with the capabilities of the Interpreter from the user's point of view, and then to explain the definition language and its syntax. We strongly suggest that you read this guide from front to back, at least the first time. Also, there may have been some changes to the software since the writing of this guide. Check the *Release Notes* for software changes before writing any important code.

First, we explain each type of object that can be defined in an application and then the application user's options when dealing with that object. The user's options are given for two reasons: so the developer can minimize the actions the user must take, and so that the developed application can be documented.

Second, we explain the method of writing object descriptions, mostly by tables and examples, and cover topics related to the UNIX system. At the end of the document are the manual pages for the built-in functions.

1 The Interpreter

Pseudo Keys

The existence of a "pseudo keyboard" with a variety of special keys is assumed in this document. It is unlikely that any terminal has all of the referenced keys. The following figure shows each of the keys discussed in this document as well as "alternate keystrokes" that will produce the same result. The ^, or caret, symbol represents the CONTROL key.

ALTERNATE KEYSTROKES FOR PSEUDO KEYS	
Pseudo key	Keystroke
SCREEN LABELED KEYS	^f1...^f8
COMMAND LINE	^z
DOWN-ARROW	^d
UP-ARROW	^u
RIGHT-ARROW	^r
LEFT-ARROW	^l
TAB	^i
BACKTAB	^t
HOME	^fb
HOME-DOWN	^fe
BEG	^b
END	^e
PREVPAGE	^v
NEXTPAGE	^w
BACKSPACE	^h
SPACEBAR	space
DEL	^x
DELETE-CHARACTER	^x
DELETE-LINE	^k
CLEAR	^y
CLEAR-LINE	^y
CLEAR-EOL	^fy
RESET	^fr
NEXT	^n
PREV	^p
PAGE-UP	^v
PAGE-DOWN	^w
SCROLL-UP	^fu
SCROLL-DOWN	^fd
INSERT-CHAR	^a
INSERT-LINE	^o
MARK	^fm

What Does FMLI Do?

The Form and Menu Language Interpreter (FMLI) is a developer tool. It recognizes a high-level "shell-like" language for defining forms, menus, and other types of frames, as well as screen-labeled keys (SLKs), a message line, a command line, and a banner. The Interpreter handles the details of frame creation, placement, navigation between frames, and processing the use of forms and menus.

Each form or menu description is stored in an ASCII file containing statements recognized by the Interpreter. Before the menu or form is displayed, the Interpreter breaks down the definition file and generates the appropriate function calls to initialize and manipulate the defined object.

There are three things you will need to know to use this tool:

- Object Architecture: which is how FMLI views the UNIX system
- How the various objects work: navigation, commands, and messages
- How to define objects: the structure and syntax of the language.

The rest of this section will deal with the first two items. The third item is covered in the chapter titled "The Definition Language."

Object Architecture

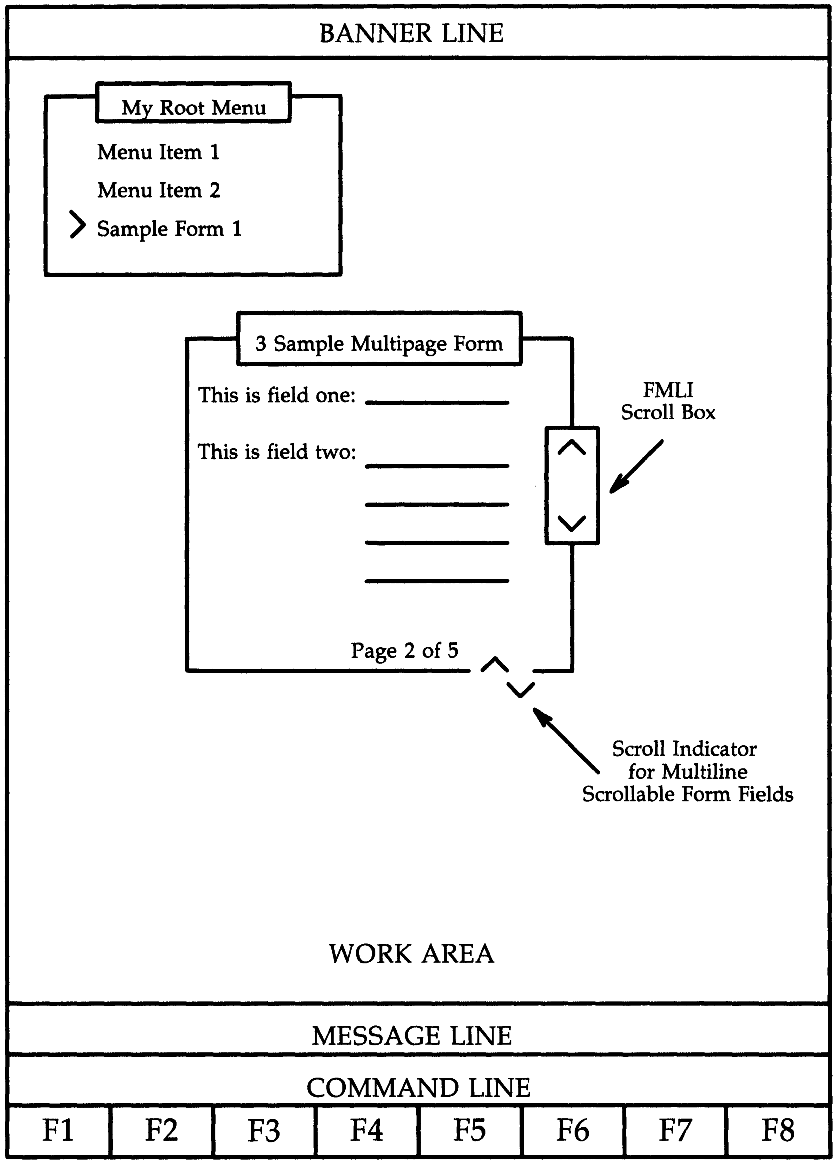
An object in FMLI is defined as a form, menu, or text frame, and the items those frames contain. When you define a form or menu, you are creating an object. The user, in such a system, needs no knowledge of UNIX system files and directories, only of objects. It is the FMLI developer's job to define objects, and operations that may be performed on those objects.

Screen Layout

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.



The screen is divided into five regions which are:

Banner Line Displays a one-line banner at the top line of the screen. The banner line is specified in the initialization file. For more information on the banner line, "working" indicator, and defining your own indicators, see "The Initialization File" in this guide, and the **indicator**(1F) manual page.

Work Area The work area is the section of the screen where "frames" are displayed. This area starts on line 2 of the screen and stops on the third line from the bottom of the terminal.

Frame A frame is an independently scrollable region of the screen surrounded by a border. FMLI allows you to define three types of frames; menus, forms, and text frames. The frame specified when FMLI is invoked is opened first. Several frames may be opened simultaneously on the display. Only one frame is the "active frame." The active frame is shown "on top" of any other frames, and its title is highlighted. The active frame may cover parts of inactive frames.

Each frame that displays three or more lines of information will contain a "scroll box" along its right frame border. A scroll box will house both an "up" indicator (^) and a "down" indicator (v). If present, the "up" or "down" indicator signals there is more information *before* or *after* the current frame, respectively. For example, the "up" indicator will appear when page two of a multiple page form is displayed.

Scroll indicators are also provided along the lower right frame border when a "multiple line scrollable form field" is active. FMLI forms are introduced later in this chapter.

Message Line The second line from the bottom of the terminal is the message line. This line is for displaying messages to the user. It is also used for one-line error and help messages. By default, the message will stay on the screen until the next key is pressed.

Command Line The command line is one line from the bottom of the terminal. The user can access this line by striking Ctrl z at which time the --> prompt appears. The user can type any command supported by the Interpreter or defined by the application. If Ctrl z is pressed while the user is in the Commands Menu, the command that is currently highlighted in that menu will appear on the command line after the prompt.

Screen-label keys (SLKs)
The bottom line of the display is reserved for the screen-labeled keys. Eight keys are displayed and associated with the eight function keys on many keyboards. There are alternate keystrokes defined if the user's keyboard does not have function keys. Each SLK has a default label and function assigned to it, depending on the type of frame active at the moment. There are two sets of SLKs. The first set can be renamed or disabled. The second set can be redefined. The code controls which set is displayed at any given time. SLKs are provided to allow the user to easily perform the functions assigned to the SLKs.

NOTE If a terminal does support hardware function keys, FMLI will use the last line of the screen to paint function key labels. The developer should be aware of this since the size of the work area will be decreased by one line to make room for these labels.

Object Operation

An object operation is a function that can be performed on an object. Object operations can be regular UNIX system commands, which the Interpreter passes to the shell for execution, but more often are either function calls built into the Interpreter, or keywords that the Interpreter handles.

The following example is a line of FMLI Definition Language code. It contains both a built-in function call and a keyword. The action to take when a selection is made from a menu is being described.

```
action='set -l MYVAR="hello"'OPEN MENU Menu.mymenu
```

In this example, OPEN is recognized as a *keyword*. Keywords are often present *outside* of backquotes. A keyword is an Interpreter command that forces an object operation to occur. In this example, the operation is OPEN and the type of object to open is a MENU. The name of the menu to open is Menu.mymenu. This example uses both the "MENU" type cast and the FMLI naming convention for menus (e.g., Menu.*). It is only necessary to use one, but you may use both. For more about the FMLI naming convention, see the introduction to Chapter 3, ''Invoking FMLI.''

Keywords

The following list of keywords is broken into two groups. The first group appears to the user in the Command Menu. The second group is primarily for the developer, but can be executed by the user from the command line. Note that some of these commands map directly to default SLKs.

User Keywords

cancel	Cancels the current command or activity. Closes an object without executing the "done" descriptor.
cleanup	Closes all objects whose lifetime is shorter than permanent.
exit	Closes all objects and exits the Interpreter.
frm-mgmt	Takes a maximum of two arguments; a keyword operation, and a frame number if the operation is move or reshape. If no arguments are given, the frame-management menu appears, and the user can select an operation which will be performed on the current frame.
goto	Makes another object current. The *goto* keyword takes as its only argument the number of a frame *or* the full path name of the object's definition file. Users should only be told about the frame number argument.
help	Invoke the action specified in the help descriptor defined for the current object.
next-frm	Moves to the next frame.
prev-frm	Moves to the previous frame.
refresh	Redraws the terminal screen.
unix	Brings up the UNIX system shell in full screen mode.
update	

Optionally takes two arguments, the first of which is a frame number or full path name. The second argument is a "true" or "false" that determines if the frame will be made current once the update is done. If the second argument is not given, FALSE is the default. Update forces the object's definition file to be re-read regardless of the presence or value of the *reread* descriptor. If there are differences between what is read and what is on the screen, the object will be re-drawn. Update will not re-read the title of an object.

NOTE The Boolean value *returned* by an FMLI command or keyword is special. It is FALSE if either the string "false" or a non–zero integer is returned, TRUE if 0 or any other string is returned. Boolean arguments *to* a command or keyword follow standard format.

Developer Keywords

open Opens an object. *open* takes two arguments. The first argument is used as a "cast," to indicate the type of object that is to be opened. The second argument is the path name of the object's definition file. For example:

```
OPEN FORM $MYOBJECTS/myform
```

Additional arguments may be added to this command. The Interpreter will pass these arguments to the opened object as described in the section "Variables."

close Closes all objects whose frame numbers appear as arguments, except those which are immortal (which means they can't be closed).

cmd-menu	Opens the Command Menu object.
nop	Does nothing. This is useful for specifying no operation for descriptors of type KEYWORD.
prevpage	Pages backward one page in the active object, if that object understands paging.
nextpage	Pages forward one page in the active object, if that object understands paging.
choices	Causes the Interpreter to check for an *rmenu* or *choicemsg* descriptor in the current field descriptor and execute it. If none exists, a message to that affect is printed.
checkworld	This routine is initiated by the SIGALRM signal. It causes FMLI to evaluate the *reread* descriptor for all objects on the screen. When the **checkworld** command is executed, the message line is cleared. If the user is reading the message when this happens they may not know why.
done	Causes the Interpreter to execute the *done* descriptor (if it exists) in an object.
mark	Marks or unmarks the current item in a multiselect menu.
reset	Resets the current field to its default value (its value when the object was opened).
togslk	Causes the Interpreter to display the set of SLKs that is not currently being displayed. It is a toggle between the two sets.

NOTE

The maximum number of arguments that may be given in an FMLI command is 25. Since an open command takes at least 2 arguments (the command is 1, the object name is 1), the maximum number of commands that can be passed to an opened object is 23. Remember, however, that once in the object you can only access the first 10 arguments (ARG0-ARG9).

What is a Form?

A form is a method for displaying and prompting for information in a frame. The form is made up of fields which are a combination of a prompt (the name of the field) and an area to enter the value of the field. A form has a title that appears on the top border, and a number to the left of the title that *identifies* the frame. As the user navigates from frame to frame, FMLI keeps a list of the identifying numbers of each frame visited. It is important to understand that the numbers have nothing to do with order. If you have five frames on the screen and you delete frame 2, the next frame you create will be assigned the number 2 because 2 is the next available number.

To the user, a form on the screen looks pretty much like a fill-in-the-blanks questionnaire. A form is a frame, and may be navigated from and to with the standard frame-to-frame navigation keys defined in the section "Frame-to-Frame Navigation."

Multipage Forms

If the form is multipage, the first page is the one that appears when the form is initially opened. Users have no way of knowing that there are more pages to the form unless you tell them. You may either put a message on the message line, or put the message in the form. To put a message in a form, equate the name descriptor to the desired message, define the page on which it should appear, but don't define an input area for that field.

Navigation Keys

Within the form, the user has the use of the following navigation keys. See the table at the beginning of this document for a list of alternate keystrokes if your terminal doesn't have these keys.

- DOWN-ARROW (^d) moves the cursor down to the next field. If you are on the last field, the cursor wraps around to the top field. If you are in a multipage form, the cursor goes to the top field on the next page of the form, if there is one. If you are on the last field of the last page of a form, the wrap is to the first field of the first page.

- UP-ARROW (^u) moves the cursor up to the previous field. If you are on the top field, the cursor wraps around to the bottom field. If you are on a multipage form, the cursor wraps to the bottom field of the previous page of the form, if there is one. If you are on the first field of the first page of a form, the wrap is to the last field of the last page.
- RIGHT-ARROW (^r) non-destructively moves the cursor right one character within a field. It does not wrap to the next field.
- LEFT-ARROW (^l) non-destructively moves the cursor left one character within a field. It does not wrap to the previous field.
- TAB (^i) moves the cursor to the next field in the form. The wrap-around feature works as it does with DOWN-ARROW.
- BACKTAB (^t) moves the cursor to the previous field in the form. The wrap-around feature works as it does with UP-ARROW.
- HOME (^fb) moves the cursor to the first character of the current field.
- HOME-DOWN (^fe) moves the cursor to the last character of the current field.
- BEG (^b) moves the cursor to the first character of the first field of the current page of a form.
- END (^e) moves the cursor to the first character of the last field of the current page of a form.
- PREVPAGE or PAGE-UP (^v) moves the cursor back one page on a multipage form if it can. It then performs a BEG.
- NEXTPAGE or PAGE-DOWN (^w) moves the cursor forward one page on a multipage form if it can. It then performs a BEG.
- BACKSPACE (^h) moves the cursor to the left, deleting the character there.
- SPACEBAR (space) replaces the current character with a space and moves the cursor one character to the right.
- DEL,DELETE-CHARACTER (^x) deletes the character under the cursor and closes the gap.
- DELETE-LINE (^k) deletes the current line of a field and closes the gap. In a single line field, it performs the same as CLEAR-LINE.

- RESET (^fr) resets a field to its default value.
- CLEAR-EOL (^fy) clears the line from the current cursor position to the end of the line.
- CLEAR, CLEAR-LINE (^y) clears the current line of the current field.

When you are editing a form, you are in the "overtype" mode. When you begin typing at the first character of a field, the field is automatically cleared. SPACEBAR, if it is the first character typed, thus appears to be clearing out the field. It is, in fact, making the space character the first character of the field, which may lead to confusion if you fail to document it.

Default SLKs

Below is a list of the default SLK keys presented with a form.

Form Default SLKs	
Key	Command
F1	HELP
F2	CHOICES/MARK
F3	SAVE/CONT/ENTER
F4	PREV-FRM
F5	NEXT-FRM
F6	CANCEL
F7	CMD-MENU
F8	CHG-KEYS

Function key 8 will default to CHG-KEYS if any of SLKs 9 through 15 are defined. For more details on SLKs, see the section titled "Additional Objects."

What is a Menu?

A menu in FMLI is a method for displaying a list of selections in a frame, determining the user's selection, and taking actions based on the selection. The menu appears as a list of items, in a left justified column, in a frame. The number of items in the menu determines the number and height of each column. If there are too many items for the menu to fit on the screen, the menu will be scrollable. If either of the scroll icons (^ for up and v for down) appears on the border of the object, the corresponding scroll key is valid. Wrapping is supported from the top to the bottom, and vice versa, in a single column menu. Wrapping is also supported from the bottom of a column to the top of the next column going to the right, and vice versa, in a multicolumn menu.

The top border of the frame has a programmer-defined name for the menu, and in the upper left is a frame number, assigned in the same manner, and used for the same purpose, as the frame number in a form (see "What is a Form?").

As the user navigates within the menu, a highlight bar shows the current item. The highlight bar is present when the cursor is used for navigating. If a character search is being done, only the characters searched are highlighted. On the left side of the bar, a > mark is also provided, in case the terminal cannot do reverse video highlighting. Navigation between frames is described in "Frame-to-Frame Navigation."

The user has two options for moving the marker to an item in a menu. The user may use the navigation keys described below, or select an item by typing its name. The user does not have to type the full name, nor worry about uppercase and lowercase. As each key is typed, the highlight bar moves to the first item in the menu that matches the total string typed so far. If the user types the letter "p," for example, the marker will move to the first item in the list that starts with the letter "p" or "P." If the user then types "r," the marker will move to the first item that starts with the letters "pr", again, irrespective of case. If the user types a letter that cannot be matched, the terminal bell will sound, or the screen will flash, depending on the terminal capability. An error message showing the string the letters have matched so far plus the unmatchable character will be printed on the message line. Once the marker has moved to an item, it may be selected by striking the Enter key. The selector bar will wrap around when it reaches the end of the menu, regardless to whether the user is moving the bar up or down.

NOTE If the user starts to type an item name, and the marker moves, and the user then changes his/her mind about what to select, the user must use one of the navigation keys before trying to use partial matching again.

Single and Multiselect Menus

For a single select menu, the user simply navigates the marker to the item to be selected and strikes a carriage return (take care to specify how this key is named in your user documentation). For a multiselect menu, the user navigates to an item to be selected and strikes the MARK SLK (function key 2) or strikes the corresponding key sequence defined in the Pseudo Key Table. Then, when the user navigates to other items, a * marker will stay beside the marked item. If you strike MARK SLK while on an item that is already marked, that item becomes unmarked. Carriage return then selects all of the marked items.

NOTE In a multiselect menu, the carriage return does not select the item the marker is currently on, unless the user has marked it with the MARK SLK. This is contradictory to the way a single select menu works, and the only way the user can tell he/she is in a multiselect menu is by the appearance of the MARK SLK. Since many users might miss this subtle difference, it would be wise to inform them of a multiselect menu with a message on screen.

Navigation Keys

The following keys are used for navigation within a menu.

- DOWN-ARROW (^d) moves the marker down one item, wrapping to the top of the next column when it reaches the bottom. If there is only one column, or the user is on the last column, the wrap is to the top of the first column.
- UP-ARROW (^u) moves the marker up one item, wrapping to the bottom of the previous column when it reaches the top of the current one. When the marker is on the first item in the menu, the wrap is to the last item in the last column.

- RIGHT-ARROW (^r) moves the marker down one item on a single column menu, right one item on a multicolumn menu. RIGHT ARROW does not wrap.
- LEFT-ARROW (^l) moves the marker up one item on a single column menu, and left one item on a multicolumn menu. LEFT-ARROW does not wrap.
- BACKSPACE (^h) is the same as LEFT-ARROW.
- SPACEBAR (space) is the same as RIGHT-ARROW.
- NEXT (^n) is the same as DOWN-ARROW.
- PREV (^p) is the same as UP-ARROW.
- HOME (^fb) moves the marker to the first item currently visible on the menu.
- HOME-DOWN (^fe) moves the marker to the last item currently visible on the menu.
- PAGE-DOWN (^w) moves the marker to the first item on the next page full of items and displays that page. If the page being paged to has fewer than 10 lines, the terminal will ring (or flash), and the page will not be displayed. The arrow keys must be used to see these items.
- PAGE-UP (^v) moves the marker to the first item in the previous page full of items and displays that page. If the page being paged to has fewer than 10 lines, the terminal will ring (or flash), and the page will not be displayed. The arrow keys must be used to see these items.
- BEG (^b) moves the marker to the first item in the menu whether it is currently visible or not, and displays the first page.
- END (^e) moves the marker to the last item in the menu whether it is currently visible or not, and displays the last page.
- SCROLL-DOWN (^fd) rolls the contents of the menu frame down one line.
- SCROLL-UP (^fu) rolls the contents of the menu frame up one line.

Default SLKs

By default, the user sees the following SLKs displayed while in a menu.

MENU DEFAULT SLKs		
Key	Menu	Multiselect
F1	HELP	HELP
F2	CHOICES	MARK
F3	SAVE	CONT/ENTER
F4	PREV-FRM	PREV-FRM
F5	NEXT-FRM	NEXT-FRM
F6	CANCEL	CANCEL
F7	CMD-MENU	CMD-MENU
F8	CHG-KEYS	CHG-KEYS

Function key 8 will default to CHG-KEYS if any of SLKs 9 through 15 are defined. For more details on SLKs, see the section titled "Additional Objects."

Additional Objects

Text Objects

Text objects are primarily used to display information to the user. Typically, the help descriptor is defined as opening a text object. While the user is in a text object, the following navigation keys are in effect.

- UP-ARROW (^u) moves the cursor up one line.
- DOWN-ARROW (^d) moves the cursor down one line.
- SCROLL-DOWN (^fd) rolls the text down one line.
- SCROLL-UP (^fu) rolls the text up one line.
- PAGE-DOWN (^w) presents the next frame full of text preserving two lines from the current frame.
- PAGE-UP (^v) presents the previous frame full of text preserving two lines from the current frame.
- BEG (^b) presents the first frame full of text.
- END (^e) presents the last frame full of text.

If either of the scroll icons (^ for up and v for down) appears on the lower right border of the object, the corresponding scroll key is valid. The Interpreter turns this capability on automatically if all of the text will not fit in the frame. In addition, any of the editing keys that can be used in a form can be used in a text object if the "edit" descriptor evaluates to TRUE.

The default SLKs presented to the user while in a text object are:

TEXT OBJECT DEFAULT SLKs	
Key	Command
F1	HELP
F2	PREVPAGE
F3	NEXTPAGE
F4	PREV-FRM
F5	NEXT-FRM
F6	CANCEL
F7	CMD-MENU
F8	blank

If any of SLKs 9 through 15 are defined, SLK 8 will be CHG-KEYS.

Though they don't appear to do anything special, text objects are still considered to be frames, and can be navigated to and from as described in "Frame-to-Frame Navigation."

Choices Menu

The CHOICES SLK maps to the **choices** command which is also available to the user. If the developer has defined a Choices Menu for a form field, executing this command will bring up that menu.

A Choices Menu is created for the user by defining the *rmenu* descriptor for a field. There are two valid methods of defining the *rmenu* descriptor. The first is to equate *rmenu* to an OPEN command. For example:

```
rmenu=OPEN MENU Menu.choices
```

where Menu.choices is a standard menu.

The second method is to equate *rmenu* to a list of choices enclosed in braces. The Interpreter will handle this list in one of two ways. If there are three or fewer items in the list, the user will use the CHOICES SLK to toggle through the list, a different choice appearing in the input field each time the SLK is pressed. If there are more than three items, the Interpreter will create a "shortterm" menu, which means that it closes as soon as the user navigates away from it. Only the CANCEL and HELP SLKs appear in this menu, and the HELP SLK applies to the form the user is in, not the Choices Menu.

The choice made by the user from a Choices Menu is automatically entered into the field to which the Choices Menu applies. Since Choices Menus are standard menus, they can be navigated to and from, but users should be warned that if they navigate from a Choices Menu, it disappears immediately.

Screen-Labeled Keys

The screen-labeled keys are provided to allow the user an easy means of performing actions that are done often. The Interpreter will provide these keys on the last line of the screen, with a label if the key has an action assigned to it. The user may strike an alternate keystroke (^f1-8) if the terminal has no function keys. The user will see the same result from striking one of these keys as by selecting that command from the Command Menu (though not all SLK commands appear in the user's Command Menu and vice versa), or striking Ctrl z and the command keyword.

CHG-KEYS will appear on SLKs 8 and 16 if the developer has defined any of the keys 9 through 15. This SLK is effectively a toggle between the two sets of SLKs.

Help

The user is presented with a SLK named HELP while in forms, menus, and text objects. Selecting HELP will bring up a frame defined by the developer. The user may or may not be able to navigate in the help frame, depending on the description set for it by the developer. Typically, the HELP frame is a developer-defined text object.

Help can also be provided to the user through the use of the CHOICES SLK or command when the user is in a form. The Choices Menu that a developer can define has already been discussed. The developer also has the option of defining a message to be printed when the user executes this command. Either with the Choices Menu or separately, the *choicemsg* descriptor can supply information on the message line. If the developer has not defined either, the message `There are no choices available` is printed on the message line when this command is executed.

Frame-to-Frame Navigation

Navigation between frames is comprised of simple moves and command actions that change the active frame. The following list defines the ways that a user can move between frames. Navigation to the UNIX system is also discussed.

- The PREV-FRM and NEXT-FRM SLKs will cause the cursor to jump from frame to frame. The frame jumped to becomes the current frame on the screen, and the frame jumped from becomes inactive. FMLI keeps a list of each frame that has been the current frame. PREV-FRM will jump to the frame number listed before the current one. NEXT-FRM jumps to the frame listed after the current one. Wrap around from the first frame listed to the last, and vice versa, occurs when these keys are used.
- Selecting FRM-MGMT from the Command Menu will bring up a Choices Menu that includes the item LIST. Selecting LIST will bring up another Choices Menu listing all opened objects. Using standard menu navigation keys, select an object and strike the carriage return. The Choices Menus disappear and the selected object becomes current. CANCEL will remove the Choices Menus and leave the user where they started.
- The CLEANUP command will close all frames not defined as "immortal," which means it cannot be closed. The last object opened that is defined as "immortal" will become the current object.
- TAB and BACKTAB will work just like NEXT-FRM and PREV-FRM, respectively, unless you are in a form, in which case these keys apply to that form's fields (see "What is a Form?").
- The *goto* command may be executed with a frame number as an argument. That frame becomes current, with the cursor on the item it was on when that frame was last active. Though this command will take a full path name argument when a developer is using it, users should only be told about frame numbers as arguments.
- The user may strike Ctrl z followed by a frame number. The action is the same as the *goto* command from the user's point of view.

- Opening an object will always cause navigation to that object.
- Closing an object will cause navigation to the frame that was active before the frame being closed was opened.

The user can invoke the UNIX system from the Command Menu or the command line. The FMLI screen will clear, and the user is in a full screen UNIX system shell. When exiting the UNIX system, a prompt message appears requesting that the user strike carriage return to continue. The FMLI screen returns in the same condition it was before the command was issued.

We have now covered the features of FMLI from the user's point of view. The next chapter describes the language used by the Interpreter to define these objects and their options.

2 The Definition Language

Prerequisites

To use the definition language, you need to be familiar with the following:

- Definition of terms, such as menu and form
- Environment variables
- UNIX system quoting mechanisms
- Command functions.

The menus, forms, and other objects are generated through the use of a definition language. This language determines how a particular object should appear and how it should be manipulated. The definition file is a simple ASCII file containing descriptors. Descriptors are the basic building blocks of the Definition Language. A descriptor defines a particular attribute that can be customized by the developer. Each of the objects that can be defined in FMLI has its own set of descriptors. Summary tables of the descriptors for each object including value type, default value, and the time at which the descriptor default is evaluated are given in the sections explaining each object. Examples of descriptions for each type of object are included.

Variables

Within a menu, text, or form object, certain characters have special meanings. These meanings are consistent with the special characters in the UNIX system shell with some additional functionality. If you are familiar with the UNIX system shell, then you know there is an "environment" which holds variables and their values. FMLI has two environments that are used for different purposes and have different capabilities.

The **set** command can set variables in a file using the *-f* option. References to these variables follow this syntax:

```
${(filename)VARIABLE}
```

where file name is a full path name and VARIABLE is the file variable name.

When a variable is expanded that does not specifically reference a file, two environments are searched. The environments are as follows:

local environment	This environment is specific to the current FMLI process. This is similar to an unexported shell variable.
UNIX system environment	The UNIX system environment is the standard UNIX system environment.

Whenever "environment" is referenced in this text, these environments are searched in the order listed.

Variables are denoted by a dollar sign ($) followed by a string. The string must be in one of the following formats:

$variable	Look for variable in the environment and expand to the value of that variable.
${variable:-default}	Look for variable in the environment and if it is found, expand to its value. If it is not found, expand to "default."

${(filename)variable}	Look for a line of the format "variable=value" in the file "filename." If such a line is found, expand to "value."
${(filename)variable:-default}	Same as above, except if variable is not found anywhere, expand to "default."

Note that file name and default may be variables, such as

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

Built-in Variables

Menu, text, and form objects may reference certain variables that have special meaning to the definition language. These variables should only be referenced, not set, within an object definition. The special variables are as follows:

$ARG*n*	This variable expands to the nth argument passed to the corresponding form, menu, or text object.
$NR	This variable expands to the number of items in the menu object.
$TEXT	This variable expands to value of the *text* descriptor within a text object.
$F*n*	This variable expands to the current value of the nth field.
$Form_Choice	This variable expands to the last choice made from a Choices Menu.
$SELECTED	This variable expands to TRUE if the current item in a multiselect menu has been marked.
$LININFO	This variable expands to null if the current menu item doesn't have a *lininfo* descriptor defined. Otherwise, it expands to the value of the *lininfo* descriptor.
$MAILCHECK	Determines the amount of time before a SIGALRM alarm automatically occurs. The minimum value for $MAIL-CHECK is 120 seconds. The default is 300 seconds.
$RET	This variable expands to the exit value of the last executable run by the Interpreter.

Syntax

Anything the Interpreter doesn't understand is ignored. This is widely encompassing. For example, a line of garbage will be ignored but so will the "selected" descriptor if a menu is single select (because "selected" has no meaning in that context). The convention of starting a comment line with the character "#" will therefore work with FMLI, except when it is nested inside quotes or backquotes. When creating a new form, menu, or text object, all quotes and backquotes must match. Quoting mismatches may cause the object to never appear, appear incorrectly, or in some circumstances cause **fmli** to core dump.

Quoting Mechanisms

If you want the special meanings of characters disabled in a string, FMLI supports a quoting mechanism similar to the UNIX system shell. Each quoting mechanism has different functions, as defined below.

- Backslash (\): A backslash causes the next single character to be taken literally.
- Single quotes (' '): Any string inside of single quotes is taken literally and as a unit. All special meanings are turned off within the quotes.
- Double quotes (" "): Double quotes group the text between them as a unit, but still allow variable expansion and the use of backquotes. Carriage returns inside double quotes are enforced.
- Backquotes (\` \`) Any command or series of commands may be enclosed in backquotes with the result that the backquoted expression expands to the output of the commands. The only character with special meanings in the output of the commands is NEWLINE. Commands may be UNIX system executables or FMLI built-in functions.

Backquotes cannot be nested (except as provided for in **regex**), but several commands may appear inside a single backquoted expression, separated by one of the following delimiters:

- semicolon (;) - Commands separated by a semicolon are executed sequentially.
- pipe (|) - When commands are separated by a pipe symbol, the output of the first command becomes the input to the second.
- AND (&&) - The meaning of command1 && command2 is run command1 and if it succeeds, then run command2.
- OR (||) - The meaning of command1 || command2 is run command1 and if it fails, run command2.

Use of Backquoted Expressions

In addition to placing backquoted expressions on descriptor lines, you can use backquoted expressions anywhere in a menu, text or form object. If a backquoted expression starts a line, it is expanded when the object is read. In this way, you can generate an entire object dynamically at run time. Thus, if a backquoted expression produces output to the message line, the expression will appear before the object being parsed is posted. For an example of a menu generated this way, see the **regex**(1F) manual page at the end of this document and "The Uses of regex" at the end of this chapter.

File Redirection

The input of a command may be redirected from a file by using " < file." Similarly, the output of a command may be sent to a file by using " > file," as in shell programming. The Interpreter does not support the shell constructs >> and 2>, which append text to a file and redirect *stderr*, respectively.

Forms

A form object is described by a series of descriptors. The first group pertains to the whole object. The second group pertains to one field of the object. This group may be repeated for additional fields. A third, optional, group that may also be repeated is the Screen-Label Key (SLK) definitions. These will be presented in the section "The Initialization File" in Chapter 3. Their usage is consistent throughout FMLI.

A form object is generally arranged like this:

> Descriptors that pertain to whole object (title, positioning, etc.)
> .
> .
> .
> Descriptors that pertain to the fields of the form object (name, default value, positioning, etc.)
> .
> .
> .
> Descriptors disabling or redefining SLKs

The programmer can define:

- the title of the form
- the screen position of the form
- the number of fields on the form
- the name of each field
- whether or not the field contains an initial value to display
- the starting position and length of each field
- whether the input is valid for each field
- whether the form is multipage or not
- new labels and functions for the SLKs.

The two tables that follow are a list of the descriptors, their default values, and at what time the default values are evaluated. The first group pertains to the entire form, and thus may only appear once in a form definition file. The second group may be reused as necessary to create additional fields.

FORM DESCRIPTORS			
DESCRIPTORS	DEFAULT IF NOT DEFINED	TYPE	DEFAULT EVALUATION TIME
form	"Form"	string	When form object is opened
help	NONE	keyword	When user asks for help
lifetime	"longterm"	string	When form object is opened
done	NONE	keyword	When user selects SAVE
init	TRUE	boolean	When form object is opened
begrow	"any"	position	When form object is opened
begcol	"any"	position	When form object is opened
close	NONE	keyword	When form object is closed
reread	FALSE	boolean	When SIGALRM occurs
altslks	FALSE	boolean	When form object is opened

Next are the descriptors that can occur once for each field in a form object.

FORM FIELD DESCRIPTORS			
DESCRIPTORS	DEFAULT IF NOT DEFINED	TYPE	DEFAULT EVALUATION TIME
name	NONE	string	When form object is opened
frow	-1 *	integer	When form object is opened
fcol	-1 *	integer	When form object is opened
nrow	-1 *	integer	When form object is opened
ncol	-1 *	integer	When form object is opened
rows	1	integer	When form object is opened
page	1	integer	When form object is opened
columns	-1 *	integer	When form object is opened
fieldmsg	NONE	string	When field is navigated to
value	NONE	string	When form object is opened
rmenu	NONE	keyword	When form object is opened
valid	TRUE	boolean	When value is changed
invalidmsg	"Input is not valid"	string	When value is changed
noecho	FALSE	boolean	When form object is opened
menuonly	FALSE	boolean	When form object is opened
show	TRUE	boolean	When form object is opened
scroll	FALSE	boolean	When form object is opened
wrap	FALSE	boolean	When form object is opened
choicemsg	NONE	string	When *choices* command is run
inactive	FALSE	boolean	When form object is opened

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

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

Following is a brief explanation of each descriptor and how it is used.

form	This is the title of the form object. The maximum length is 74 characters. If it exceeds this, it will be truncated to 74 characters.
help	If the user asks for help while in this form, this command will be run.
lifetime	This determines when this form will be removed from the screen. Acceptable values and when they allow an object to close are:

- shortterm	when another object becomes current
- longterm	when CLOSE or CLEANUP is issued
- permanent	when a CLOSE is issued
- immortal	cannot be closed

done	When the user selects the SAVE SLK, this descriptor's value is executed. If this descriptor is not defined, the form is simply closed.
init	If it's value evaluates to FALSE, then the form will not be posted.
begrow, begcol	These descriptors determine the offset of the *top left* corner of the object (begrow=0, begcol=0 is the upper left corner of the FMLI Work Area). These descriptors are of the type "position," which means that in addition to integral values, the following are acceptable:

center	will be centered
current	will overlap current frame
distinct	will *not* overlap current object (if possible)
any	positioned with least amount of total overlap

The values presented above can be assigned to *begrow* or *begcol* independently to force a restriction on the row or column only. If integral values are supplied and either *begrow* or *begcol* are outside the screen boundary, a default value of "any" will be given to the erroneous descriptor.

close	This is expanded and executed when the user selects CANCEL.
reread	If the descriptor's expanded value is TRUE, the form will be updated by rereading it's description file. A SIGALRM alarm occurs every $MAILCHECK seconds (see the section *Built-in Variables*). When it occurs, all objects whose *reread* descriptor evaluates to TRUE will be updated. The "title" descriptor is not re-read. Execution of *reread* causes the message line to clear.
altslks	If this descriptor appears, or expands to TRUE, SLKs 9 through 16 are displayed when the object is initially opened. If the descriptor does not appear, or evaluates to FALSE, SLKs 1 through 8 are displayed.

Here are the descriptors that can occur once for each field in a form. If you define more than one, only the last one will be used.

name	This keyword begins the description of a new field in the form object. The field name is used to tell the user what piece of information is wanted in a certain field. It can also be used to put a message in a form.
nrow, ncol	Take integer arguments. They position the name in the frame. If the value is negative, the name will not be posted. If the integer is too large (the position is off the screen), the entire form is not displayed.
value	This is the default value for the input field.
frow, fcol	These position the input field in the frame. If either value is negative, then the input field will not be displayed.
rows, columns	The maximum size of the input field. Generally, they describe the length and width of the region in which the users can type.
page	Denotes which page of a form this field will be on. The description may evaluate to an integer, or the strings "*" or "all," which place the field on all pages of the form. By default, all fields will appear on one page (i.e., page = 1).

choicemsg	Defines a message to be put on the message line when the user selects CHOICES.

rmenu

This is used to specify a list of choices, delimited by spaces, for a particular input field. There are two formats that are acceptable. The first is a list of choices, separated by spaces, enclosed in braces. The spaces after the opening brace and before the closing one are mandatory.

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

The line **rmenu=**{} is illegal syntactically and will cause a core dump. The minimum requirement is **rmenu=**{ " " }. If this list has three or fewer items, the user toggles through the items by striking the CHOICES SLK. Four or more items will appear in a Choices Menu. The user's selection is automatically placed in the local variable Form_Choice, the value of which is inserted into the active field when the Choices Menu closes.

The second acceptable format is an open command. The descriptor evaluates to opening a menu, and the user selects from that menu. The action associated with each choice in that menu must set the local variable Form_Choice. That value is inserted into the active field when the menu closes. If there are four or more choices, a menu will be presented and the item selected from the menu will be placed in the field. You can force a Choices Menu by the line **rmenu=OPEN MENU menuname**.

There must be at least one active field in a form. If you open a form with only one field defined, and that field cannot be posted because *rows* or *columns* is negative or 0, **fmli** will core dump.

valid

If this evaluates to FALSE, the current value input by the user is invalid. Checking the validity of the field is often done by evaluating a backquoted expression. The built-in function *regex* is often used for field validation. Validation is performed when a field is "visited" (navigated to by the user). As a warning, fields that are never visited will not be validated.

For example, a user visits a frame with five fields. The first two fields are visited and the last three are not. Therefore, only the first two fields will be validated when the user leaves the frame.

FMLI does not validate all of the fields before saving it. Thus it is possible for an illegal value to sneak in (e.g., the initial contents of a field are set by a variable that gets corrupted). FMLI validates the current field whenever its value changes. It also validates the current field when a **save** operation is performed. If necessary, one can validate all fields as part of the *done* descriptor, which is always evaluated when the SAVE key is pressed.

invalidmsg	This string is printed on the message line when the input for this field is invalid.
noecho	If this does not evaluate to FALSE, then when the user types in this field, what the user types will not be echoed on the input field (often used for passwords).
menuonly	If this descriptor is set to TRUE, then the only acceptable input for this field is one of the choices in *rmenu*.
show	If this evaluates to FALSE, then the field will not be shown. Note that if the field is not shown, it still counts as a field for the purpose of expanding the variable $Fn.
scroll	If this is not set to FALSE, then the input field can be scrolled. This means that the input field can be as long as the entry the user types.
wrap	If this evaluates to FALSE, then the cursor will not automatically wrap to the next input line when the user is typing an entry in this input field.
inactive	If this descriptor evaluates to TRUE, the item is displayed in the form, but cannot be navigated to. The default is TRUE if the descriptor is there but not defined. If the descriptor is not there, the field will be active.
fieldmsg	This string will appear on the message line when this field is navigated to.

A feature of the Interpreter is the ability to expand the value of any descriptor. Typically, the name descriptor in a form object would be a literal string. However, you can let the name field be a calculated value. For example, suppose you want the value of a field to be the value of an environment variable called *$MYNAME*. It is legal to say `value=$MYNAME`. When the form object is read, the value of that field will be the expansion of the variable *MYNAME*. In fact, each time any field value changes in this form, the variable will be re-expanded.

In some cases, this approach may cause inefficiency. Consequently, two "casts" are provided to control this; *const* and *vary*. If these directives are used, they must appear after the equal sign (=) on the descriptor line. For example,

```
    value=const $MYNAME
```

would define value as whatever the variable *MYNAME* expanded to when the object is opened. The value will never be expanded again as long as the object remains opened, even if a *reread* is issued. The **update** command will always cause re-expansion.

If the directive *const* appears, the field will only be expanded once. If the directive *vary* appears, then the descriptor will be re-evaluated each time the object changes. This is useful for descriptors that are normally evaluated only once, such as name.

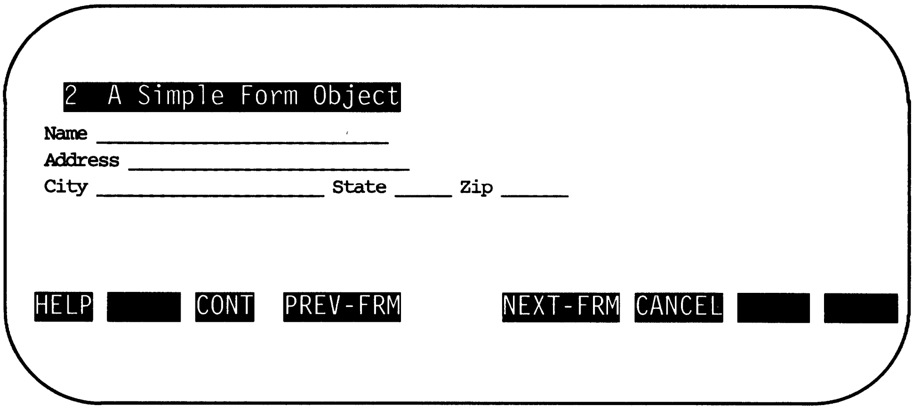
The use of *const* can make a form, menu, or text operation more efficient, especially when the value of a descriptor is a backquoted expression which calls UNIX system commands.

The *const* keyword should be used with caution because the assumption here is that this descriptor value is always constant and never needs to be re-evaluated.

On the next page is an example of a simple form. If we call it Form.simp, the user could open this form by any action that evaluates to:

```
    OPEN FORM Form.simp
```

The system reads the ASCII file, constructs an internal representation of how the form will appear to the user, expands the value descriptors for each field, and displays the resulting form.



```
 2  A Simple Form Object
Name ________________________
Address ___________________________
City ____________________ State _____ Zip ______



HELP        CONT  PREV-FRM          NEXT-FRM CANCEL
```

This is the Description Language code used to generate this form, followed by an explanation of what the user could and could not do in this form.

```
form="A Simple Form Object"
done=`set -l NAME="$F1" -l ADDR="$F2" -l CITY="$F3" -l STATE="$F4" -l ZIP="$F5"`

name="Name"
nrow=1
ncol=1
frow=1
fcol=6
rows=1
columns=20
valid=`regex -v "$F1" '^[A-Za-z,. ]+$'`
value=const "$NAME"

name="Address"
nrow=2
ncol=1
frow=2
fcol=9
rows=1
columns=28
value=const "$ADDR"

name="City"
nrow=3
```

continued

```
ncol=1
frow=3
fcol=6
rows=1
columns=14
value=const "$CITY"
rmenu=OPEN MENU $MYPATH/Menu.city "$STATE"

name="State"
nrow=3
ncol=21
frow=3
fcol=27
rows=1
columns=2
value=const "$STATE"
rmenu={ NY NJ CT CA IL ME TX }
menuonly=true

name="Zip"
nrow=3
ncol=31
frow=3
fcol=35
rows=1
columns=5
value=const "$ZIP"
valid=`regex -v "$F5" '[0-9]{5}'`
```

The descriptor *form* gives the form a name other than the default "Form." The descriptor *done* tells the Interpreter what to do when the user selects the SAVE SLK; set the environment variable NAME to the current value of field 1, set ADDR to the value of field 2, etc. These values would then appear in the form by default the next time this form is opened.

Next, the descriptors that can be used repeatedly define five fields where data will be input. The first field here puts up the string "Name" in row 1, column 1 of the form, with an input field starting on the same row at column 6. The input field is 1 row high and 20 columns long. It has a default value of whatever is stored in the environment variable $NAME. The validity of the input to this field is checked using the built-in function **regex**. The expression makes sure that the name is all letters, spaces and commas.

The other fields; Address, City, State, and Zip, are all defined in the same manner, but of these, only Zip has a validation description. The validation for zip code makes sure that the user enters exactly five digits.

The City field gives an example of a command style rmenu. Menu.city could list possible city choices pertaining to the current state. Menu.city would make use of the argument $STATE which is passed to it. The action descriptor for each choice would set Form_Choice and close the menu.

The state field gives another example of the *rmenu*. In this case, if the user asks for CHOICES, a Choices Menu will display which will give the available two-letter state codes. Because the *menuonly* descriptor is given, only these choices are legal.

The user may edit the form using various editing keys described in the first part of this document. Suppose the user changes the value of the NAME field. When the user strikes RETURN or TAB to exit the NAME field, the valid descriptor is expanded. In order to expand this, the system runs the internal command **regex** and attempts to match the value of the NAME field against the pattern "^[A-Za-z,.]+$" -- in other words, one or more letters, commas, periods, or spaces. (See the **regex**(1F) built-in function manual page.)

Assuming the user has entered valid information in all the fields, the user may strike the SAVE SLK. This causes the *done* descriptor to be evaluated, and again a backquoted expression is encountered, containing the internal command SET. This command sets 5 variables in the user's environment to the values of each of the 5 fields. It then closes the form.

Menus

A menu object has a series of descriptors that pertain to the whole object followed by a series of descriptors that will pertain to each item in the menu or each Screen-Label Key. The menu description is arranged like this.

Descriptors that pertain to whole object (title, positioning, etc.)

.

.

.

Descriptors that pertain to menu object lines (name, description, action, etc.)

.

.

.

Descriptors that pertain to Screen-Label Keys (name, button number, action, etc.)

The programmer has control of the following options in a menu:

- single or multiselection menu
- opening the menu with specific items already selected
- placement of the menu on the screen
- the lifetime of the menu
- whether or not to show a specific choice
- the action to take for each item
- the action to take when the menu is closed.

The following table shows the descriptors used to describe a menu.

MENU DESCRIPTORS			
DESCRIPTOR	DEFAULT IF NOT DEFINED	TYPE	DEFAULT EVALUATION TIME
menu	"Menu"	string	When menu object is opened
multiselect	FALSE	boolean	When menu object is opened
help	NONE	keyword	When user asks for help
lifetime	"longterm"	string	When menu object is opened
init	TRUE	boolean	When menu object is opened
begrow	"any"	position	When menu object is opened
begcol	"any"	position	When menu object is opened
close	NONE	keyword	When menu object is closed for any reason
reread	FALSE	boolean	When timer goes off
done	NONE	keyword	Upon RETURN in a multiselect menu
altslks	FALSE	boolean	When object is opened

Next are the fields that can occur once for each item in a menu. The set of descriptors describing each item must start with the "name" descriptor.

MENU ITEM DESCRIPTORS			
DESCRIPTOR	DEFAULT IF NOT DEFINED	TYPE	DEFAULT EVALUATION TIME
name	NONE	string	When menu object is opened
description	NONE	string	When menu object is opened
action	NONE	keyword	When this line or button is selected
lininfo	NONE	string	When this line is selected
show	TRUE	boolean	When menu object is opened
selected	FALSE	boolean	When menu is opened.
itemmsg	NONE	string	When item is navigated to

Following is a brief description of each descriptor and how it is used.

menu	This is the title of the menu object. It will be truncated to 45 characters.
multiselect	Tells the Interpreter that this is a multiselect menu. SLK 2 will map to the MARK command, and the "action" descriptor is ignored for all selections.
help	If the user asks for help within this menu object, this command will be run (see Definitions).
lifetime	This determines when this menu object will be removed. The acceptable values are: shortterm - closes whenever another object becomes the current object longterm - closes when the user issues a CLEANUP or CLOSE command permanent - closes whenever the user issues a CLOSE command immortal - cannot be closed
init	If this expands to FALSE, the menu object will not be opened; otherwise it will.
begrow, begcol	These descriptors describe the position of the menu object's top left corner. Values can be one of the following: *center* - menu will be centered *current* - as close to the current frame's position as possible *distinct* - as far from the current frame's position as possible *any* - system chooses a position to minimize overlap *integer* - an absolute position. Causes frame to appear in same position. If begrow and begcol force the menu to display off the screen, they will default to "any."

close	This is expanded when the user closes or cancels the menu.
reread	When the SIGALRM alarm occurs, the *reread* description is expanded. If it expands to TRUE, the menu will be reread.
done	Evaluated when the user strikes carriage return in a multi-select menu. Ignored in a single select menu.
altslks	If the item's expanded value is not FALSE, SLKs 9 through 16 are displayed when the object is initially opened. The default, if the descriptor is not used, is FALSE, which displays SLKs 1 through 8.

Here are the descriptors that can occur once for each item in a menu.

name	The string that will appear in the menu.
description	This will be the part of the line displayed but not highlighted when the user is on this line.
action	The value is a string, equivalent to a command that could be typed on the command line. Multiple backquoted expressions are allowed, as they are with any descriptor, but the final value of this descriptor must be a single keyword expression. This descriptor is ignored if the menu is multiselect.
lininfo	When the user selects this menu item, this descriptor's string value will be put into the local environment variable LININFO. If it is not defined, LININFO will be null. Also, when the *getitems* function is executed, if this string is defined, its value will be substituted for the item's "name" string.
show	This determines whether this menu item should be displayed. It will not be displayed if the value is FALSE.
selected	This descriptor determines whether a menu item should default to selected (TRUE) or nonselected (FALSE) when the menu is opened. The default is FALSE.

itemmsg	This string is displayed on the message line when this item is navigated to.

In addition, SLKs may be defined in a menu description file.

A menu object usually starts with the line `menu=title`. The default value for title is "Menu." In order to create a menu, you would use a series of descriptors, which are the building blocks of the definition language. Each descriptor defines a particular alternative or function of the menu. Descriptors are in the format `descriptor=value`. When grouped together, these descriptors determine how the object will appear to the user and how it can be manipulated. Following is a simple menu description file.

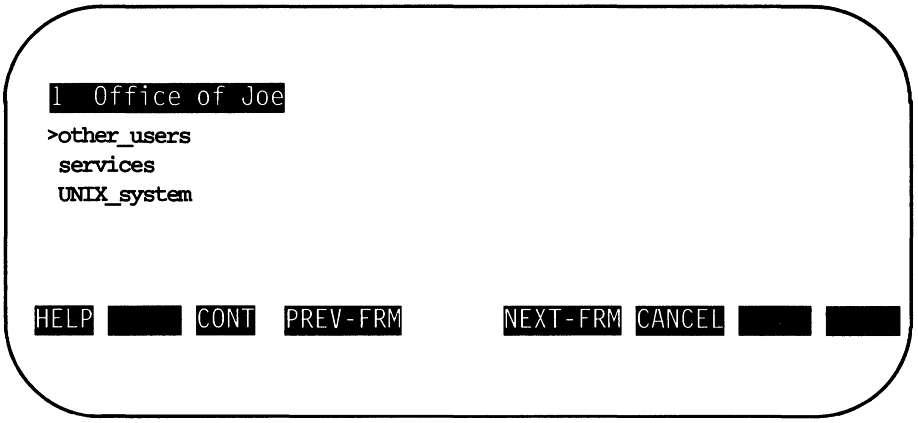
```
menu=Office of $LOGNAME

name=other_users
action=OPEN MENU $MYOBJECTS/Menu.users

name=services
action=OPEN MENU $MYOBJECTS/Menu.serve

name=UNIX_system
action=unix
```

Here is how a simple menu description file would be displayed to the user.



```
1  Office of Joe
>other_users
 services
 UNIX_system



HELP        CONT   PREV-FRM          NEXT-FRM CANCEL
```

When the user selects one of these items, the corresponding action is executed. In this example menu, you are using three types of descriptors to generate the title, menu item names, and the action to take when an item is selected. Single-instance descriptions within a menu definition are used to generate attributes that refer to the entire menu, in this case, the title of the menu is defined with:

```
menu="Office of $LOGNAME"
```

Note that the environment variable LOGNAME is expanded by the Interpreter and included as part of the definition string.

Multi-instance descriptors are used to generate attributes for each item on a menu; in this case, *name* and *action* describing each of the three items in the menu.

In a menu, if the combined length of the name and description of an item is greater than 76 characters, the next item defined will not be posted.

Text Objects

A text object has a series of descriptors that pertain to the whole object followed by a series of descriptors that will pertain to the Screen-Label Keys. A text object description is generally arranged like this:

> Descriptors that pertain to whole object (title, text, positioning, etc.)
>
> .
>
> .
>
> .
>
> Descriptors that pertain to Screen-Label Keys (name, button number, action, etc.)

A text object usually starts with the line `title=title`. The default value for title is `Text object`. Following is a table of descriptors, default values, and default evaluation times.

TEXT OBJECT DESCRIPTORS			
DESCRIPTORS	DEFAULT IF NOT DEFINED	TYPE	DEFAULT EVALUATION TIME
title	"Text"	string	When text object is opened
text	" "	string	When text object is opened
edit	FALSE	boolean	When text object is opened
wrap	TRUE	boolean	When text object is opened
rows	10	integer	When text object is opened
columns	30	integer	When text object is opened
help	NONE	keyword	When user asks for help
lifetime	"longterm"	string	When text object is opened
done	NONE	keyword	When the object is closed
init	TRUE	boolean	When text object is opened
begrow	"any"	position	When text object is opened
begcol	"any"	position	When text object is opened
close	NONE	keyword	When text object is opened
reread	FALSE	boolean	When text object is opened
altslks	FALSE	boolean	When text object is opened

Following is a brief description of each descriptor and how it is used.

title	This is the title of the text object. It will be truncated to 75 characters.
help	If the user asks for help on this text object, this command will be run.

lifetime	This determines when this object will be removed from the screen. Acceptable values and when they allow an object to close are: - shortterm when another object becomes current - longterm when CLOSE or CLEANUP is issued - permanent when a CLOSE is issued - immortal cannot be closed
done	When the user selects CANCEL, this descriptor is evaluated. If it expands to FALSE, the text object stays open; otherwise the object is closed.
init	If its value evaluates to FALSE, then the object will not be posted.
begrow, begcol	These descriptors determine the offset of the *top left* corner of the object (begrow=0, begcol=0 is the upper left corner of the FMLI Work Area). In addition to integral values, the following are acceptable: *center* - will be centered *current* - will overlap current frame *distinct*- will *not* overlap current object (if possible) *any* - positioned with least amount of total overlap The values presented above can be assigned to *begrow* or *begcol* independently to force a restriction on the row or column only. If integral values are supplied and either *begrow* or *begcol* is outside the screen boundary, a default value of "any" will be given to the erroneous descriptor.
close	This is expanded and executed if the user selects CANCEL.
reread	When the SIGALRM alarm occurs, if this descriptor evaluates to TRUE, the text object will be reread and redrawn.
altslks	If the item's expanded value is TRUE, SLKs 9 through 16 are displayed when the object is initially opened. If this descriptor is not defined, or expands to FALSE, SLKs 1 through 8 are displayed.

rows, columns	These should be set to the number of rows high and columns wide you want the frame to be.
text	This descriptor should evaluate to the text you want to display.
edit	If this descriptor evaluates to TRUE, then the user can modify the text. Otherwise, the text is read only.
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.

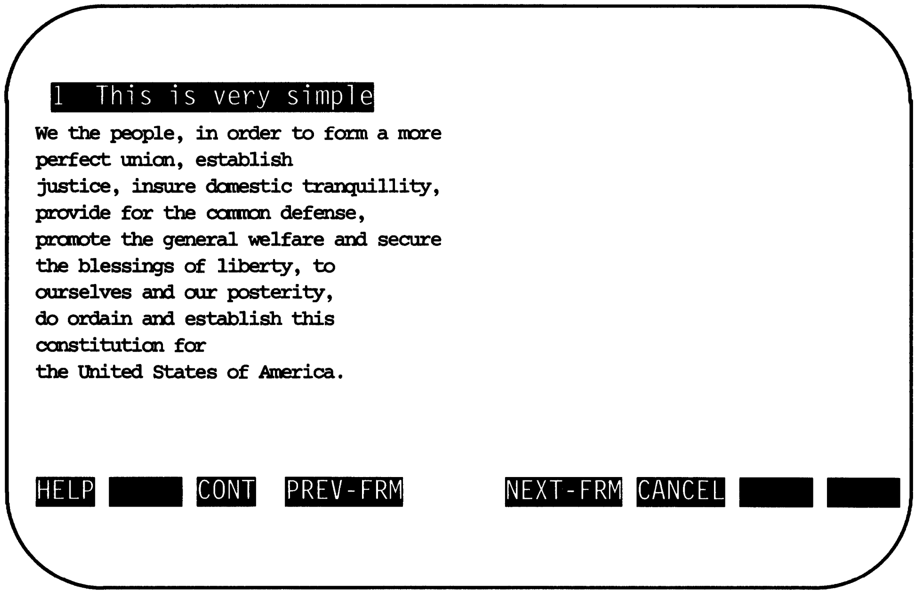
In addition, the descriptors for SLKs may be included in the text object description. They do not vary from their use in other objects.

Here is a simple description file for a text object.

```
title="This is very simple"
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."
```

The object would look like this:



```
1  This is very simple
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.
```

Notice that even though the text is wrapped at 40 columns, the original new-line characters are preserved.

A more interesting way to do this would be:

```
title="This is less simple"
lifetime=longterm
text="`readfile $ARG1`"
columns=`longline`
```

This example illustrates the use of arguments that may be passed to menu, text, or form objects. You don't have to write a separate text object for each file that is to be displayed. Instead you pass $ARG1 to the object when you open it. For example, if this object were opened by a line in a menu that looked like this:

```
action=OPEN TEXT $MYOBJECTS/Text.standard help1
```

$ARG1 would expand to "help1", that file would be read by the built-in function *readfile,* and all of the text 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 the section *Variables* and the *readfile* manual page.

Built-in Functions

Using the backquotes will allow FMLI to recognize built-in commands as part of an object definition or as part of an action definition. Built-in functions are handled internally by the Interpreter and invoke no process when executed.

When a backquoted expression produces output, this output is considered part of the descriptor. Care must be taken that this does not produce an illegal value on the descriptor line. For instance if MYVAR is set to "hello," then:

```
        action=`echo $MYVAR`OPEN MENU Mymenu
```

will be equivalent to:

```
        action=helloOPEN MENU Mymenu
```

This will produce an illegal descriptor value since `helloOPEN` is not a known keyword.

Below is a list of the FMLI built-in functions. There are manual pages for each at the end of this document.

echo	The **echo** command outputs its operands.
indicator	The **indicator** command allows you to control the "working" indicator and bell, and allows you to define your own indicators on the banner line.
message	The **message** command outputs its operands to the Interpreter message line. The bell can also be controlled. To force a message to be "permanent," use the **message** option *-p*, which will display the message until another one is displayed. When that message clears, the "permanent" message will reappear. To clear a "permanent" message, use the *-p* option and a null string.
pathconv	The **pathconv** command converts an alias to a full path name. It can also produce a shortened version suitable for use as a form title.
readfile, longline	The **readfile** command reads the file passed as its argument and writes it to standard output. 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** command 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** command performs regular expression matching on its string input (utilizing **regex**(3X)).
run	The **run** command is used to invoke an executable in full screen mode.
set,unset	These commands set and unset environment variables either in the UNIX system environment or in files.
shell	The **shell** command is used to run a command using the UNIX system shell. This is useful for performing tasks that are not provided by the language (for example, the UNIX system **test** command or **sed**(1)).
getitems	The **getitems** command takes as its only argument a delimiter string. It returns a list of the currently selected items, separated by the delimiter supplied.
reinit	The **reinit** command takes as an argument the name of an initialization file. It is used to make changes to the FMLI while staying in the current application.
setcolor	The **setcolor** command allows you to redefine an existing color, or define new colors if your terminal allows more than the eight colors already defined in FMLI.
getfrm	The **getfrm** command returns the current frame number. It takes no arguments.
fmlcut	The **fmlcut**(1F) command is used to cut out selected fields of each line of a file.
fmlgrep	The **fmlgrep**(1F) command is used to search a file for a certain pattern.

For more information about how these commands are used, see the section "Syntax."

Co-processing Commands

Five other built-ins allow an object or several objects (that is, form, menu, or text) to communicate to an external process through a pipe. The Interpreter would 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 functions are as follows:

cocreate	Initializes communication to a process using named pipes.
cosend	Sends strings from the Interpreter to the process. The -n option performs a "no wait" condition that sends text, but doesn't block for a response.
cocheck	Checks the incoming pipe for information. Returns TRUE or FALSE.
coreceive	Performs a "no wait" read on the pipe. Takes a process ID as an argument.
codestroy	Terminates this communication

Co-processing

In addition to the built-in commands, the Interpreter can execute UNIX system programs and UNIX system shell commands via constructs in the Form and Menu Definition Language. Both built-ins and UNIX system commands are specified using the backquoting mechanism described earlier in this section. If a command is recognized as a built-in command, it is executed by the Interpreter (i.e., no process invocation is necessary); otherwise, it is passed to the shell for execution.

The restriction here is that these commands do not require any "interaction" with the user. In other words, these commands run to completion without user confirmation, or prompting. If an application wishes to execute a UNIX system program that does require some sort of interaction during its execution, the Interpreter provides two mechanisms:

- The Interpreter could "suspend" the frames that are displayed and execute the process in full screen. The built-in function *run* supports this capability. For example, the expression

  ```
  `run my_word_processor`
  ```

  would instruct the Interpreter to clear the screen and execute the word processing application in full screen. Once the user exits from the word processor, the Interpreter will resume where it left off, restoring the screen to its presuspended state.

- The second alternative is a more "integrated" one. It allows a process to communicate with the user via an object (menu, text, form). To support this capability, the Interpreter provides a feature called co-processing. A co-process does not have direct access to the terminal screen. It communicates with the Interpreter. The Interpreter then posts the messages in the object that contains the co-processing descriptors.

The co-processing feature is made up of five built-in commands: **cocreate, cosend, cocheck, coreceive**, and **codestroy**, which support inter-process communication.

The **cocreate** command is responsible for initializing the process and setting up pipes between the Interpreter and the co-process. The **codestroy** command is responsible for cleaning up when the communication has been completed. The built-in **cosend** is used to send information to the co-process via the pipe and block for some response by the co-process. The -n option to

cosend performs a "no wait" condition. This means that **cosend** will send information to the co-process but will not block for a response. The **cocheck** command will check the "incoming" pipe for information. The **coreceive** command will perform a "no-wait" read on the pipe. The purpose of these built-in functions is to provide a flexible means of "interaction" between the Interpreter and a co-process; to be responsive to asynchronous activity.

It is important to note that information passed to the Interpreter from a co-process is treated as *text* only. Commands (for example, OPEN, CLOSE, UPDATE) will not be recognized by the Interpreter.

To illustrate the use of enhanced co-processing, consider a UNIX system program that wishes to "talk" to the user as it executes (interactive program). The following is a sample menu which displays the item "talk." When selected, the operation specified by the "action" descriptor will post an "interactive" form as defined by Form.talk.

```
menu="My Menu"

name="talk"
action=OPEN FORM Form.talk
```

In the object "Form.talk" shown below:

- The *close* descriptor will be responsible for destroying the communication.
- The *reread* descriptor will check the pipe and "reread" the object definition if there is information pending.
- The backquoted expression will create the co-process when the form is opened (while the descriptors are being parsed, before the form is posted).
- Field 1 will be an "inactive field" used simply to display text received from the co-process.

- Field 2 will be 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 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 an object is closed).

```
form="Talking ..."
close='codestroy MYPROC'
reread='cocheck MYPROC'

'cocreate -i MYPROC $MYSTUFF/bin/talk'

name=""
frow=0
fcol=0
rows=5
columns=20
inactive
value="'coreceive MYPROC'"

name=""
frow=5
fcol=0
rows=1
columns=20
valid='cosend -n MYPROC "$F2"'TRUE

name=abort
button=8
action='message "Communication stopped ..."'close
```

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" -eq "goodbye" ]
   then
      break
   fi
done
echo "goodbye"
vsig
```

The supplied executable **vsig** is used to send a signal telling the Interpreter that information is pending. This interrupt causes *reread* to execute. The **vsig** executable is documented in the **vsig**(1F) manual page. For more information about co-processing, see the **coproc**(1F) manual page.

The Uses of regex

The **regex** command is an FMLI built-in that is useful in a variety of situations. It takes as input a stream of text and compares each line of text against one or more "patterns." These "patterns" represent regular expressions that are provided as arguments to the **regex** command line. A "template" must appear after each pattern on the command line (exceptions to this rule will be discussed later). A "template" is a string that is written to **stdout** if the corresponding pattern is matched. The **regex** command will always write the template of the *first* pattern which is matched.

Consider the following example:

```
`cat /etc/passwd | regex 'cat' 'dog' 'open' 'close' 'up' 'down'`
```

This **regex** statement contains three pattern/template combinations which will be compared against each line of text in the file "/etc/passwd." The patterns are "cat," "open," and "up." For every line of text that matches one of these patterns, **regex** will write the appropriate template to **stdout**. The templates are "dog," "close," and "down," respectively.

The **regex** command also provides nine "registers" to save "pieces" of a pattern for use in the template.

```
`cat /etc/passwd | regex '^(J[a-zA-Z]*)$0:' '$m0'`
```

The variables "$0" and "$m0" denote **regex** "register" references in the pattern and template, respectively. This statement tells **regex** to match any line in */etc/passwd* that begins with a "J" and contains only alphabetic characters. The portion of the pattern which is surrounded by parentheses is then placed in register 0. The contents of register 0 are then referenced in the template (via $m0). Though only one register is used in this example, **regex** statements can contain up to 9 register references. The purpose of this statement is to write only the first field of the matched pattern (delimited by ":") to **stdout**.

Before proceeding, it is important to note that information can be passed to/from UNIX system executables and FMLI built-in commands. Try creating a menu item with the following action line:

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

Redirecting statements to the message line via the *message* built-in is a useful debugging aid.

NOTE The "NOP" keyword means do nothing (no operation). It is used because the terminal will beep if a keyword is not present.

Field Validation

The most popular use of **regex** is for form field validation. As stated previously, if a pattern is matched by **regex**, **regex** will write the corresponding template to **stdout**. The **regex** command will also return the value TRUE which, for FMLI built-ins, 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-ins, "FALSE" is equivalent to a UNIX system command that exits with a non-zero exit status, for example, the following form field validation function (i.e., *value* descriptor definition).

```
        valid=`regex -v "$F1" '^[a-zA-Z0-9]*$'`
```

The "-v" option tells **regex** to use the argument that follows (rather than **stdin**) 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.

The **regex** command will return TRUE if the current value of field 1 consists entirely of alphanumeric characters and will return FALSE otherwise. Since no template exists, **regex** will NOT write to **stdout**.

Generating Dynamic Objects

One of the most powerful uses of **regex** is for generating objects dynamically. Consider the problem of designing a menu that displays the contents of a UNIX system directory. If a user selects a file from the menu, FMLI should display the contents of the file in a text frame. For simplicity, assume the directory contains only ASCII files.

```
`ls "$ARG1" | regex '^(.*)$0$' '
              name="$m0"
              action=OPEN TEXT Text.display "$m0"'`
```

When the menu is first initialized, FMLI will run **ls(1)** on the passed argument to retrieve the names of all files residing under "$ARG1." The output of **ls** is then passed to the FMLI built-in **regex**. Since the single **regex** pattern matches every line of input, **regex** will return the template *for every file in the directory*. The result is a series of template expansions that generate an FMLI menu definition. Notice that the name of the file (referenced by "$m0") is passed as an argument to Text.display.

NOTE In FMLI, backquoted expressions that appear by themselves (rather than as part of a descriptor value) are evaluated *when the object is first initialized.*

An object need not be entirely dynamic nor entirely static. Parts of an object may be generated via backquoted expressions, and parts may be generated using conventional means.

A Case Statement

If the *-e* option is present, **regex** will **evaluate** the template of the matched pattern before passing it to *stdout.*

For example, consider the following menu item definition:

```
name="Say hello"
action=`echo 0 | regex '0' '`message hello`'`nop
```

This definition will not produce the message "hello" when the menu item is selected. The **regex** command will simply pass the template string *as is* to *stdout.* On the other hand, the definition

```
name="Say hello"
action=`echo 0 | regex -e '0' '`message hello`'`nop
```

will produce the desired result.

For a more realistic example, consider the UNIX system based program *is_he_there.* This program takes as its only argument the name of a person to be searched for in a data base. The program *is_he_there* will return 0 if the person exists and 1 if the person does not exist in the data base. Given this introduction, consider the following *done* descriptor definition. The descriptor *done* is specific to an FMLI form and is evaluated when a user attempts to SAVE a form.

```
done=`message "Looking for \"$F1\"";
   is_he_there;
   regex -e -v "$RET"
   '0'    '`message "Found \"$F1\"";
            set -l KEYCMD="OPEN FORM Form.foundhim \"$F1\""
          `'
   '1'    '`message "Sorry, can not find \"$F1\"";
            set -l KEYCMD="NOP"
          `'
   `$KEYCMD
```

Since the *done* descriptor expects a keyword value, the local variable KEYCMD is used to construct the keyword command string. The keyword command "NOP" simply means do nothing. The variable RET is an FMLI variable that expands to the exit status of the most recently executed UNIX system process. Notice that nested double quotes are "escaped" using the character "\", a convention that is consistent with the UNIX system shell language.

Fmlcut(1) and Fmlgrep(1)

As mentioned previously, FMLI provides a set of built-in functions that are available to an application developer. These function provide capabilities such as regular expression matching (**regex**), variable setting/unsetting (**set**, **unset**), and message line output (**message**). Two built-ins which are quite useful, primarily in form processing, are **fmlcut** and **fmlgrep**.

The built-in **fmlcut** will retrieve selected fields from a file. The **fmlgrep** built-in will search a file for a given pattern. FMLI developers who are also UNIX shell programmers should be quite familiar with both of these commands.

Consider the following FMLI form definition that displays selected entries from the file /etc/passwd. When the form is first initialized, **fmlgrep** will retrieve the entry for login-id "$ARG1" from /etc/passwd. The variable "ARG1" refers to the first argument passed to the form. The output of **fmlgrep** is then piped to the FMLI built-in **set**.

The *value* of each form field is determined by "cutting" the contents of variable *tmpvar* accordingly. The *-f* option to **fmlcut** specifies the field number, the *-d* option specifies the delimiter character.

NOTE

The string "const" is provided with each field value to indicate that all defaults are "constant." That is, they need only be evaluated when the form is initialized. "Const" should not be used if a field's default value may vary. For example, "const" should not be used if a field value depends on the current value of another field (e.g., value="$F1").

```
`fmlgrep "$ARG1" /etc/passwd | set -l tmpvar`

name="login id"
nrow=1
ncol=1
rows=1
columns=20
frow=1
fcol=10
value=const "`echo $tmpvar | fmlcut -f1 -d:`"

name="passwd"
nrow=2
ncol=1
rows=1
columns=20
frow=2
fcol=10
value=const "`echo $tmpvar | fmlcut -f2 -d:`"

name="user id"
nrow=3
ncol=1
rows=1
columns=20
frow=3
fcol=10
value=const "`echo $tmpvar | fmlcut -f3 -d:`"

name="group id"
nrow=4
ncol=1
rows=1
columns=20
frow=4
fcol=10
value=const "`echo $tmpvar | fmlcut -f4 -d:`"
```

Shell vs. run

When FMLI encounters a command within backquotes, it will first determine whether or not it is a known built-in command. If it is, FMLI will execute the command internally (without creating a new UNIX system process). If the command is not a built-in, FMLI will generate a new process to execute the command.

Consider the following:

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

In this example **date** is not an FMLI built-in command yet **message** is. FMLI will generate a new process to execute the UNIX system command **date**(1) and pass its output to the FMLI built-in **message**.

The following is a description of two FMLI built-in commands (**shell** and **run**) that do invoke UNIX system processes.

shell

The **shell** built-in command will execute a UNIX system command *without* clearing the screen. It takes as its only argument the string to be passed to the UNIX system shell for execution. This command is used primarily to create "in-line" UNIX system shell scripts. Since FMLI is not a full implementation of the UNIX system shell, the **shell** built-in allows developers to take full advantage of the UNIX system shell language from within the context of the FMLI language.

For example, consider the following menu item definition:

```
    name="Do something"
    action=`message "About to execute a shell script";
            shell "
                if [ -d "$ARG1" ]
                then
                        echo "It is a directory";
                else
                        echo "It is not a directory";
                fi
            " | message`nop
```

Notice that the output of the **shell** built-in is piped to the FMLI **message** built-in.

run

The **run** built-in allows one to execute a UNIX system command *in full screen.* This is useful for executing UNIX system commands/applications which are "interactive." When the **run** command is encountered FMLI will clear the screen, suspending the "frames" that are posted, and executed the UNIX system command/application in full screen. Once the UNIX system command has terminated, the "frames" will be restored as they were before the command was executed. Note however that *stdin, stdout,* and *stderr* are not modified by the **run** command, thus, no information can be piped to/from **run**.

The following is a menu item definition that contains the *run* built-in:

```
name="Edit my .profile"
action=`message "About to edit your .profile ...";
            run vi .profile;
            message "Returning to FMLI"`nop
```

NOTE: The command string to the *run* command should not be surrounded by quotes.

If one wishes to use the UNIX system shell as an "intermediary," consider the following:

```
action=`run sh -c "vi .profile"`nop
```

where **sh** is the UNIX system shell command. The *-c* option to **sh** tells the UNIX system shell to execute the command string which follows. This is useful for trapping any UNIX system shell messages that may be produced such as illegal command strings, unknown commands, etc..

3 Invoking FMLI

Introduction

The executable file **fmli** requires at least one argument, the initial object to open. Subsequent interactions are driven by this initial object. Optionally, you may provide the names of an initialization file, a commands file, and an alias file. The initialization file provides specific global instructions that allow for customization of the application, such as screen colors and default SLKs. The commands file allows the definition of commands specific to that application. The alias file allows you to define short, easy to use, aliases for long path names to devices and files.

The generalized command for invoking the Interpreter is:

fmli [-i <initialization file>] [-c <commands file>] [-a <alias file>] <file> [<file>...]

where <file> is the full path name of the file describing the object to be opened initially, and must follow the naming convention Menu.xxx for a menu, Form.xxx for a form, and Text.xxx for a text file, where xxx is any string that conforms to UNIX system naming conventions. The descriptor *lifetime* will be ignored for all frames opened by argument to fmli. These frames have a lifetime of "immortal" by default.

NOTE

FMLI does not use the EOF marker to determine when to exit an application, it uses the **exit** command. Thus, it is strongly advised that input to FMLI or FMLI applications not be from a pipe (|), a redirected file (<), or a here document (<<).

The Initialization File

One of the arguments you may give when invoking **fmli** is the name of an initialization file. In the initialization file, an application developer is able to specify the following:

- A short-term introductory object displaying the application name
- A banner, its position, and other objects on the banner line
- Color attributes for all objects
- Screen-label keys (SLKs) and their layout.

Each is described in detail below.

The Introductory Object

This object is displayed briefly when the application starts, and is then cleared from the screen and replaced by the frame(s) you specify as the initial object(s). The introductory object is specified by using four of the descriptors normally used to define a text object. Those descriptors are shown in the table below. Note that when not specified in the initialization file, title and text default to NULL. If both title and text descriptors are missing from the initialization file, no introductory object is displayed.

INTRODUCTORY OBJECT DESCRIPTORS			
DESCRIPTOR	DEFAULT IF NOT DEFINED	TYPE	DEFAULT EVALUATION TIME
title	""	string	At initialization time
text	""	string	At initialization time
rows	10	position	At initialization time
columns	50	position	At initialization time

This is a subset of the complete list of descriptors for text objects, and more about each of these descriptors can be found in the section "Text Objects." Please note, however, that the defaults for "rows" and "columns" shown above only apply in the initialization file. The color of the introductory object is controlled by the descriptors described under the section "Color Attributes."

The syntax for this object is simple and can be seen in the following example:

```
title="WELCOME TO"
text="My Application
Copyright (c) 1988
My Software, Inc.
All rights reserved."
rows=5
columns=25
```

Backquoted expressions, containing calls to functions built-in to the Interpreter, may also be used, as in this line

```
text="`readfile myintrotext`"
```

which will cause the text file **myintrotext** to be read and passed to the text descriptor as the argument. More about quoted and backquoted expressions is explained in the section titled "Syntax."

The Banner

The application can display a banner on the banner line if you include at least the first of the following descriptors.

BANNER DESCRIPTORS			
DESCRIPTOR	DEFAULT IF NOT DEFINED	TYPE	DEFAULT EVALUATION TIME
banner	NONE	string	When init file is read
bancol	"center"	position	When init file is read
working	"Working"	string	When init file is read

See the description of *begrow* in the section "Forms" in this guide for an explanation of the type "position."

The following lines, in an initialization file, would give you a banner with the program name and the date on the banner (top) line of the screen starting in the 30th column.

```
banner="MYPROGRAM - `date`"
bancol=30
```

The working indicator appears right justified on the banner line. Taking care that other items on the banner do not run into this area is the responsibility of the developer. Changing the working indicator, to BUSY for example, is done by defining the "working" descriptor in your initialization file:

```
working="BUSY"
```

To disable the working indicator entirely:

```
working=""
```

You may also put an application specific indicator on the banner line by using the built-in function *indicator*, which is documented in the FMLI built-in function manual pages.

Color Attributes

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. The defined color descriptors for both text and background are:

- black
- blue
- green
- cyan
- red
- magenta
- yellow
- white.

You may redefine these descriptors, or add new ones, with the *setcolor* built-in function, which is documented in the manual pages.

The following language descriptors are used to specify color attributes for various screen entities. If the terminal does not support color, these descriptors are ignored.

screen	Color of the screen (screen background).
banner_text	Defines the color of all text on the banner line. The background for this text is defined by *screen*. Failing to define banner_text will cause the banner text to default to white.
window_text	Color of the text in a frame (text foreground). The text background color will be the screen background color (see the *screen* descriptor).

active_border	Color of the frame borders when a frame is current (border foreground). This will enforce the "solid line" look of the borders. The border background color will be the screen background color. Note that 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 is the case, set `highlight_bar=green` and `highlight_bar_text=red` to produce the proper color combination.
inactive_border	Color of the frame borders when a frame is non-current (border foreground). Once again the border background color will be the screen background color.
active_title_text	Color of the title text when a frame is current (title foreground).
active_title_bar	The title background color when a frame is current (title background).
inactive_title_text	Color of the title text when a frame is non-current (title foreground).
inactive_title_bar	The title background color when a frame is non-current (title background).
highlight_bar	Color of the menu selector bar (bar background).
highlight_bar_text	Color of the bar text (bar foreground).
slk_text	Color of the screen-label key text.
slk_bar	Color of the screen-label keys (background for slk_text).

All of these descriptors are of type STRING and accept the color values given in the discussion of the banner line.

Due to the nature of **curses**(3X), colors must be set in pairs. This means you must set both the foreground and background for an area of the screen, otherwise it will default to monochrome.

NOTE If you set the foreground and background to the same color you will not be able to see the text.

The built-in function *setcolor* allows you to define your own colors, if the terminal is capable of it. If you want to write machine independent code that uses the setcolor capability, use the "or" operator in your backquoted expression. For example:

```
screen=`setcolor blue 100 24 300 || echo blue`
```

will set the screen to the default blue if the new one can't be defined. Of course, if this terminal can't display color, the Interpreter automatically defaults to monochrome. The color descriptors are allowed only in the initialization file. They will be ignored in other files.

If you reinitialize the FMLI application with a new background color, text in the banner line will be shifted one character to the right. To avoid this problem, force the banner to be reevaluated by including the banner descriptors in the new initialization file.

Screen-Label Keys

Screen-Label Keys (SLKs) appear at the bottom of the Interpreter screen and provide easy access to a number of widely used functions. They are analogous to a set of menu items that are always displayed and can be selected at any time. There are 8 SLKs that map directly to the 8 function keys that appear on a majority of terminals (alternative escape sequences are listed in the Pseudo Keys Table.)

By default, the Interpreter provides 2 levels of SLKs. There are 8 SLKs that appear at the first level and an alternate set of 8 SLKs that appear at the second level. The Interpreter has only defined the first set for each object type. These defaults were given in the first part of this document. If you define SLKs 9 through 15 in the second set, the eighth and sixteenth SLKs default to CHG-KEYS, which serves simply as a toggle to flip-flop between levels.

SLKs 1 through 7 in the first set can be disabled, but not redefined. SLKs 8 through 16 may be redefined. However, if you define SLK 8 or 16, the user looses the ability to toggle between the two sets of SLKs. Redefining the SLKs can be done in the initialization file, in which case they become the defaults. They may also be defined in form, menu, and text files, in which case they override the defaults while that object is active.

The developer can define which set of SLKs first appears when the object is opened by setting the single instance descriptor *altslks.* If this descriptor evaluates to TRUE, SLKs 9 through 16 will be displayed when the object is first opened. *altslks* can appear in form, menu, and text descriptions.

The following is a list of multi-instance descriptors that can be used to redefine the Screen-Label Keys (SLKs).

name	Name that is displayed on the SLK. Must be 8 or fewer characters. Defining the name as a null string (`name=""`) will disable the SLK.
action	Operation to perform when the particular SLK is selected.
button	The value of this descriptor is the number of the function key (1 through 16) to which the SLK refers.

show	If its value expands to FALSE, then the SLK will not appear.
slk_layout	Describes the layout of the screen-labeled keys on the screen. Two groupings are supported; "4-4" and "3-2-3." The default, if this descriptor is not used, is "3-2-3".

The following is an example of how an application developer could use an initialization file to disable F7 (CMD-MENU) and define F9 (the first SLK in set 2) as the EXIT key:

```
name=""
button=7
show=true (this line is optional)
name="Exit"
button=9
action=exit
show=true (this line is optional)
```

NOTE

The SLKs must be the last thing defined in any descriptor file.

Modifying Command Keywords

Keywords can be added to the Command Menu or disabled. This is done by creating a command file and supplying it as an argument when **fmli** is invoked. There is an absolute maximum of 64 command keywords. The format for adding or disabling is as follows:

```
name=<cmd name>
action=<action to take>
help=<keyword operation>
```

To add a new command, for example,

```
name="date"
action=`date | message`NOP
help=OPEN TEXT $MYOBJECTS/Text.datehelp
```

will allow a user to have a **date** command that puts the date on the message line.

To disable an existing command, for example, *frm-mgmt,*

```
name="frm-mgmt"
action=NOP
```

To change the "default" help for an FMLI command,

```
name="frm-mgmt"
help=OPEN TEXT $MYOBJECT/Text.frmhelp
```

NOTE It is important to remember that the **exit** command is the only way to exit an FMLI application. Thus, if you disable the **exit** command, you must make sure that somewhere in your application code there is a menu item with the **action** descriptor defined as the **exit** command. Otherwise, the user will be stuck in your application with no way to exit.

The contents of the command file will be reflected in the Command Menu. One should avoid keywords that are a partial match of another keyword such as "cr" which is a partial match of "create."

Adding Path Aliases

The developer can define a path alias to simplify references to objects or devices with lengthy path names. Whenever a path name is referenced that does not begin with a "/" or a "$" the Interpreter 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 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:/usr/spool/uucppublic
```

would search *$HOME/myfiles* first, and if the file is not found, search */usr/spool/uucppublic* whenever the alias *MYFILES* is used. This is similar to the way *$PATH* is searched in the UNIX system. The alias file is specified to the Interpreter with the *-a* option during invocation.

Terminal Independence

FMLI uses the UNIX system terminfo data base to determine the terminal's capabilities. The default path to this database is **/usr/lib/terminfo** if the environment variable $TERMINFO is not set. New terminals not described in this database can be added to the terminfo under the proper sub-directory named by the first character in the terminal's name. For example, the 5425 terminal description would be in $TERMINFO/5/5425.

In your **.profile**, after TERM is set, you should execute a `tput init` command to initialize **curses**.

In some cases, if an **stty sane** is done, **stty tab3** would be necessary to ensure a sane screen. (Borders of OBJECTS may be distorted.) This happens on the color console.

4 FMLI Manual Pages

FMLI Manual Pages

The following are the manual pages for the FMLI built-in functions and executables. The functions included are:

- chkuser
- coproc
- echo
- fmlcut
- fmlgrep
- getfrm
- getitems
- indicator
- message
- pathconv
- readfile, longline
- regex
- reinit
- reset
- run
- set, unset
- setcolor
- shell
- vsig

NOTE The **chkuser** command is not a built-in command of FMLI. But it is used as an executable for the system administration application for UNIX System V/386 Release 3.2.

NAME

chkuser – run program for privileged users

SYNOPSIS

chkuser [**-u**] [**-c** command]

DESCRIPTION

The *chkuser* command is a "setuid-to-root" program that will pass root privileges down through the *exec* system call. This allows users to issue commands that might take root permissions only.

The **-u** option will validate the user and set the return code to zero if the user has administrative access or to one if the user is not a privileged user. See example 1.

The **-c** option uses the *exec* system call to invoke the underlying software that will actually perform the administrative function. The command to *exec* is checked against the list of legal commands. The command and its arguments to .chkuser have to be enclosed in double quotes. See example 2.

NOTE: The *chkuser* command is not a built-in command of FMLI. But it is used as an executable for the System Administration application for UNIX System V/386 Release 3.2.

EXAMPLES

```
(1) action=OPEN `/usr/vmsys/admin/.chkuser -u| regex
        '^0$' "$VMSYS/OBJECTS/mailset/Menu.mail"
        '.*' "$VMSYS/OBJECTS/mailset/Text/malpriv"`

(2) /usr/vmsys/admin/.chkuser -c "command ARG1 ARG2"
```

SEE ALSO

/usr/vmsys/admsets/base-adm

NAME

coproc: cocreate, cosend, cocheck, coreceive, codestroy – communicate to a process

SYNOPSIS

cocreate [**-r** rpath] [**-w** wpath] [**-i** id] [**-R** refname]
        [**-s** send_string]
        [**-e** expect_string] command

cosend [**-n**] id string

cocheck id

coreceive id

codestroy [**-R** refname] id

DESCRIPTION

The *cocreate* command initializes communication to a process using named pipes. This means that the process will expect strings on its input and send information on its output.

The *cosend* command works two ways. With the **-n** option, *cosend* does not wait for a response. The process should use *vsig* to force the strings into the pipe and then signal that it wishes to send. This causes a reread to occur in the current frame. The *vsig* executable is supplied on the FMLI disk, and is described in detail on the *vsig*(1F) manual page in chapter 4 of the *FMLI Programmer's Guide.*

The *cocheck* command should be called from a reread descriptor. The default value of one of the fields in the form should include the *coreceive.*

Without the **-n** option, *send_string* and *expect_string* are used to tell when input and output are completed on the pipe. In other words, the Interpreter during a *cosend* will output all the strings given as arguments followed by *send_string,* to say that it is through giving information. Then it will read all the output from the process until it sees the *expect_string.* By default, the Interpreter will send no *send_string* and expect no *expect_string* (it will expect only one line of output from the process). Read the warning below if you use *cosend* without the **-n** option.

The *codestroy* command should usually be given the **-R** option, since you may have more than one process with the same name, and you do not want to kill the wrong one. It keeps track of the number of *refnames* you have assigned, and when the last one is killed, kills the process (*id*) for you.

The *id* is used to refer to the process. If none is specified, the name of the process is used.

Refname is a "local" name for a process. This is useful when multiple objects reference the same process (i.e., when multiple objects perform a *cocreate* on the same process). Thus, when a *codestroy* operation is performed you will usually want to destroy only the local reference to the process rather than the entire pipe.

The *rpath* tells *cocreate* what file to use to read information from. The *wpath* tells *cocreate* what file to use to write information to. These files are usually used for processes that naturally write to a certain pipe or for having one process talk to many different Interpreters. If *rpath* and *wpath* are not specified, paths will be picked in **$HOME/tmp**.

Command should be a program followed by its arguments.

Here is some advice for writing these programs. If this program is to be written in "C", make sure to flush output after writing to the pipe (a good way to check this is to run *cat* | *prog* | *cat* from shell). As of this writing, *awk*(1) and *sed*(1) can not be used because they do not flush after lines of output. Shell scripts are well-mannered, but slow. "C" is recommended. If possible, use the default *send_string, rpath* and *wpath*. In most cases, the *expect_string* will have to be specified. (Note: the *expect_string* need only be the initial part of the line, and there must be a new-line at the end of the output). *Id*'s are usually used when the same process is used with different options and different meanings.

Codestroy will usually work best in "close=" lines in menus and forms. The "close=" is guaranteed to be evaluated when a window is closed.

EXAMPLE

```
        .
        .
        .
init='cocreate -i BIGPROCESS initialize'
close='codestroy BIGPROCESS'
        .
        .
        .
reread='cocheck BIGPROCESS'

name='cosend -n BIGPROCESS field1'
        .
        .
        .
name="Receive field"
inactive=TRUE
value='coreceive BIGPROCESS'
```

WARNING

If *cosend* is used without the **-n** option, a coprocess that does not answer will cause the Interpreter to permanently hang.

SEE ALSO

awk(1), cat(1), sed(1).

NAME

echo – put string on virtual output

SYNOPSIS

echo [string] . . .

DESCRIPTION

If no argument is given, *echo* looks to *stdin* for input. *Echo* directs each string it is passed to *stdout*. It is often used in conditional execution or for passing a string to another command.

EXAMPLES

Validate Field 1 as integer:

```
valid=`echo "$F1" | regex '^[0-9]+$'`
```

Write information to *stdout* when a form is done:

```
done=`echo "$LOGNAME is on-line" | message`
```

SEE ALSO

echo(1).

NAME

fmlcut – fmlcut out selected fields of each line of a file

SYNOPSIS

fmlcut **-c**list [file ...]
fmlcut **-f**list [**-d**char] [**-s**] [file ...]

DESCRIPTION

Use *fmlcut* to fmlcut out columns from a table or fields from each line of a file; in data base parlance, it implements the projection of a relation. The fields as specified by *list* can be fixed length, i.e., character positions as on a punched card (**-c** option) or the length can vary from line to line and be marked with a field delimiter character like *tab* (**-f** option). *fmlcut* can be used as a filter; if no files are given, the standard input is used. In addition, a file name of "–" explicitly refers to standard input.

The meanings of the options are:

list	A comma-separated list of integer field numbers (in increasing order), with optional **-** to indicate ranges [e.g., **1,4,7**; **1-3,8**; **-5,10** (short for **1-5,10**); or **3-** (short for third through last field)].
-c*list*	The *list* following **-c** (no space) specifies character positions (e.g., **-c1-72** would pass the first 72 characters of each line).
-f*list*	The *list* following **-f** is a list of fields assumed to be separated in the file by a delimiter character (see **-d**); e.g., **-f1,7** copies the first and seventh field only. Lines with no field delimiters will be passed through intact (useful for table subheadings), unless **-s** is specified.
-d*char*	The character following **-d** is the field delimiter (**-f** option only). Default is *tab*. Space or other characters with special meaning to the shell must be quoted.
-s	Suppresses lines with no delimiter characters in case of **-f** option. Unless specified, lines with no delimiters will be passed through untouched.

Either the **-c** or **-f** option must be specified.

EXAMPLES

fmlcut –d: –f1,5 /etc/passwd	mapping of user IDs to names
\`who am i ǀ fmlcut –f1 –d" "\`	to get the current login name.

DIAGNOSTICS

The following error messages may be displayed on the FMLI message line:

ERROR: line too long	A line can have no more than 1023 characters or fields, or there is no new-line character.
ERROR: bad list for c / f option	Missing **-c** or **-f** option or incorrectly specified *list*. No error occurs if a line has fewer fields than the *list* calls for.

ERROR: no fields The *list* is empty.

ERROR: no delimiter Missing *char* on **-d** option.

ERROR: cannot handle multiple adjacent backspaces
Adjacent backspaces cannot be processed correctly.

WARNING: cannot open <filename>
Either *filename* cannot be read or does not exist. If multiple file names are present, processing continues.

SEE ALSO

fmlgrep(1F).

NAME

fmlgrep – search a file for a pattern

SYNOPSIS

fmlgrep [options] limited regular expression [file ...]

DESCRIPTION

The *fmlgrep* command searches files for a pattern and prints all lines that contain that pattern. The *fmlgrep* command uses limited regular expressions (expressions that have string values that use a subset of the possible alphanumeric and special characters) like those used with *ed*(1) to match the patterns. It uses a compact non-deterministic algorithm.

Be careful not to use FMLI special characters (e.g., $, `, ', ") in the *limited regular expression*. It is safest to enclose the entire *limited regular expression* in single quotes '...'.

If no files are specified, *fmlgrep* assumes standard input. Normally, each line found is copied to standard output. The file name is printed before each line found if there is more than one input file.

Command line options are:

-b	Precede each line by the block number on which it was found. This can be useful in locating block numbers by context (first block is 0).
-c	Print only a count of the lines that contain the pattern.
-i	Ignore upper/lower case distinction during comparisons.
-h	Prevents the name of the file containing the matching line from being appended to that line. Used when searching multiple files.
-l	Print the names of files with matching lines once, separated by newlines. Does not repeat the names of files when the pattern is found more than once.
-n	Precede each line by its line number in the file (first line is 1).
-s	Suppress error messages about nonexistent or unreadable files.
-v	Print all lines except those that contain the pattern.

SEE ALSO

fmlcut(1F).

DIAGNOSTICS

Return value is TRUE if any matches are found, FALSE if otherwise.

BUGS

Lines are limited to BUFSIZ characters; longer lines are truncated. BUFSIZ is defined in **/usr/include/stdio.h**.

If there is a line with embedded nulls, *fmlgrep* will only match up to the first null; if it matches, it will print the entire line.

NAME

getfrm – returns the current frame number

SYNOPSIS

getfrm

DESCRIPTION

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

EXAMPLE

```
        action=OPEN TEXT stdtext `getfrm`
```

The action here is to open the text object "stdtext," passing the current frame number to that text object as its ARG1.

NAME

getitems – returns a list of the currently marked menu items.

SYNOPSIS

getitems [delimiter_string]

DESCRIPTION

The *getitems* command takes a delimiter string as its only argument. It returns a list of the names (or *lininfo,* if it is defined) of the currently marked menu items, delimited by the argument string. If no argument is given, the default delimiter is NEWLINE.

EXAMPLE

This code defines a menu:

```
Menu="Example"
multiselect=TRUE
done=`getitems ":" | message`

name="Item 1"
action=`message "You selected item 1"`

name="Item 2"
lininfo="This is item 2"
action=`message "You selected item 2"`

name="Item 3"
action=`message "You selected item 3"`
```

If all three items in this menu were marked, when he RETURN key is pressed the following string will appear on the message line:

```
Item 1:This is item 2:Item 3
```

Note that since *lininfo* is defined for item 2, its value is substituted for the name.

NAME

indicator – display application specific alarms and/or the "working" indicator

SYNOPSIS

indicator [**-c** column] [**-l** length] [**-o**] [**-w**] [**-b** [n]] "[string]" ...

DESCRIPTION

The **-c** option dictates what column of the banner line to start the indicator string on. Column is an integer from 0 to 79. If the **-c** option is not used, the default is 0.

The **-l** option limits the length of the indicator. If the string is longer than length, it will be truncated. Length is an integer from 1 to 80. If **-l** is not used, the default is the entire string.

The **-o** option causes indicator to "tee" its output to *stdout.*

The **-w** option turns on the "working" indicator.

The **-b** option rings the terminal bell *n* times, where *n* is an integer from 1 to 10. The default value is 1. If the terminal has no bell, the screen is flashed instead, if possible.

If the *indicator* command is being used solely for the bell or working indicator control, remember to give it a null string argument unless input is being piped to it. The string should always be the last argument given. The indicator is not automatically cleared.

EXAMPLES

When the value entered in the field is wrong, ring the bell three times and put up an indicator saying WRONG in column 1.

```
invalidmsg=`indicator -b 3 -c 1 "WRONG"`
```

To clear the indicator after telling the user the entry is wrong:

```
invalidmsg=`indicator -b 9 -c 1 "WRONG";
     indicator -c 1 "      "`
```

NAME

message – puts its arguments on message line

SYNOPSIS

message [-**t**] [-**p**] [-**o**] [-**b** [n]] "[string]"

DESCRIPTION

The *message* command puts its string arguments out onto the message line. If there is no string, the *stdin* input to *message* will be used. If the **-t** flag is set, the message is output in temporary form (it will be removed after the next key is pressed). This is the default argument. If the **-p** flag is set, the message is output in permanent form. This argument is used for prompts, it will stay up until the next message is put up. The **-o** flag forces *message* to "tee" its message to *stdout*. The **-b** [*num*], where *num* is an integer from 1 to 10, rings the terminal bell *n* times. The default value is 1. If the terminal has no bell, the screen is flashed instead, if possible.

If the *message* command is being used solely for the bell or working indicator control, remember to give it a null *string* argument unless input is being piped to it. The *string* should always be the last argument.

EXAMPLES

When the value entered in the field is wrong, ring the bell 3 times and then put up the invalid field message "Try again!"

```
invalidmsg=`message -b 3 ""`Try again!
```

Put out a message to tell the user what is being done:

```
done=`message hello has been set in your
environment`
```

NAME

pathconv – search Interpreter criteria for filename

SYNOPSIS

pathconv [**-v** alias] [**-f**] [**-t**]

DESCRIPTION

The *pathconv* command is used to convert an alias to its pathname. It takes the alias as a string from *stdin*. The **-v** option allows the alias to be embedded in the command string. The **-t** option implies that *pathconv* should expand the alias in a format suitable for display as a frame title. This format is a shortened version of the full pathname, created by deleting components of the path from the middle of the string until it is under 42 characters in length, and then inserting "..." between the remaining pieces. The **-f** option means return the full path (this is the default).

EXAMPLES

Here is a menu that is titled using *pathconv*:

```
menu=`pathconv -t -v $ARG1`
        .
        .
        .
```

This will result in the same thing:

```
Menu=`echo $ARG1 | pathconv -t`
        .
        .
        .
```

SEE ALSO

echo(1F).

NAME

readfile, longline – reads file and gets longest line

SYNOPSIS

readfile file

longline [file]

DESCRIPTION

The *readfile* command reads the file named in its argument. No translation of new-lines is done. It keeps track of the longest line it reads and if there is a subsequent call to *longline*, the length of that line, including the newline character, is returned. The *longline* command can be given an argument instead, in which case it will calculate its longest line.

EXAMPLES

Here is a typical use of *readfile* and *longline* in a text object:

```
        .
        .
        .
text="`readfile myfile`"
columns=`longline`
        .
        .
        .
```

DIAGNOSTICS

If the file does not exist, *readfile* will return FALSE (i.e., the expression will have an error return).

SEE ALSO

cat(1).

NAME

regex – match patterns against a string

SYNOPSIS

regex [**-e**] [pattern template]...pattern [template]

regex [**-e**] **-v** "string" [pattern template]...pattern [template]

DESCRIPTION

The *regex* command takes a string (from *stdin*, or supplied with **-v**) and a list of **pattern/template** pairs, and runs *regex*(3X) on the string versus each of the patterns until there is a match. When a match occurs, it writes the corresponding **template** to *stdout* and returns TRUE. The last (or only) pattern does not need a template. If that is the pattern that matches the string, the function simply returns TRUE. If no match is found, *regex* returns FALSE.

The **-e** option tells the function to evaluate the corresponding template and write the result to *stdout*.

The patterns are regular expressions of the form described in *regex*(3X). In most cases the pattern should be enclosed in single quotes to turn off special meanings of characters.

The **template** may contain the strings $m0 through $m9, which will be expanded to the part of the *pattern* enclosed in (...)$0 through (...)$9 constructs (see examples below). Note that if you use this feature, you must be sure to enclose the **template** in single quotes so that the Interpreter doesn't expand the $m0 through $m9 variables at parse time. This feature gives *regex* much of the power of *cut*(1), *paste*(1), and *grep*(1), and some of the capabilities of *sed*(1). If there is no **template**, the default is "$m0$m1$m2$m3$m4$m5$m6$m7$m8$m9". Note that only the final **pattern** may lack a **template**.

EXAMPLES

To "cut" the 4th through 8th letters out of a string:

```
`regex -v "my string is nice" '^.{3}(.{5})$0'
'$m0'`
```

In a form, for validating input as an integer:

```
valid=`regex -v "$F5" '^[0-9]+$'`
```

In a form, to translate an environment variable which contains one of the numbers 1, 2, 3, 4, 5 to the letters a, b, c, d, e:

```
value=`regex -v "$VAR1" 1 a 2 b 3 c 4 d 5 e
'.*' 'Error'`
```

Note the use of the pattern '.*' to mean "anything else".

In the example below, all three lines constitute a single backquoted expression. This expression, by itself, could be put in a menu descriptor file. Since backquoted expressions are expanded as they are parsed, and output from a backquoted expression (the *cat* command, in this example) becomes part of the descriptor file being parsed, this expression would read **/etc/passwd** and make a virtual menu of all the login ids on the system.

```
`cat /etc/passwd | regex '^([^:]*)$0.*$' '
name=$m0
action=`message "$m0 is a user"`'`
```

DIAGNOSTICS

If none of the patterns match, *regex* returns FALSE, otherwise TRUE.

WARNING

Patterns and templates must often be enclosed in single quotes to turn off the special meanings of characters. Especially if you use the $m0 through $m9 variables in the template, since the Interpreter will expand the variables (usually to " ") before *regex* even sees them.

SEE ALSO

cut(1), grep(1), paste(1), sed(1) in the *User/System Administrator's Reference Manual.*

regcmp(3) in the *Programmer's Reference Manual.*

BUGS

The regular expressions accepted by *regcmp* differ slightly from other utilities (i.e., sed, grep, awk, ed, etc.).

Regex with the **-e** option forces subsequent commands to be ignored. In other words if a backquoted statement appears as

```
`regex -e ...; command1; command2`
```

command1 and *command2* would never be executed. However, dividing the expression into two

```
`regex -e ...``command1; command2`
```

would yield the desired result.

NAME

reinit – runs an initialization file

SYNOPSIS

reinit filename

DESCRIPTION

The *reinit* command takes an initialization **filename** as its only argument. The Interpreter will parse and execute this file, and then continue running the current application. Typically used to change the defaults set by the initialization file that was named when *fmli* was invoked.

The *reinit* command does not re-display the introductory object or change the screen label keys layout.

NAME

reset – reset the current form field to its default values

SYNOPSIS

reset

DESCRIPTION

The *reset* command resets a field in a form to its default value; i.e., the value displayed when the form was first opened.

NAME

run – run an executable

SYNOPSIS

run [**-s**] [**-e**] [**-n**] [**-t** title] program

DESCRIPTION

The *run* command runs a program, using the PATH variable to find it. The **-s** option means "silent", implying that the screen will not have to be repainted when this is done. The **-e** option means to prompt the user before returning to the Interpreter only if there is an error condition (by default the user is always prompted). The **-n** means never prompt the user (useful for programs like **vi** which the user must do some specific action to exit in the first place). The **-t** option gives the name this process will have in the pop-up menu generated by the *frm-list* command. This option implies the ability to suspend the UNIX system process and return to the FMLI application.

EXAMPLE

Here is a menu that uses *run*:

```
menu="Edit special System files"

name="Password file"
action=`run -e vi /etc/passwd`

name="Group file"
action=`run -e vi /etc/group`

name="My .profile"
action=`run -n vi $HOME/.profile`
```

NAME

set, unset – set and unset environment variables in core or in files

SYNOPSIS

set –<l ¦ ffilename ¦ e variable=value> . . .

unset –<l ¦ ffilename variable> . . .

DESCRIPTION

The *set* command can be used to set variables in the environment or add variables to environment-like files. The *unset* command removes these variables. There are two built-in environments; a local one, and the UNIX system environment which passes variables between processes. These environments are accessed by the **-l** and **-e** options, respectively. When expanding variables, the Interpreter checks the local environment first, and then the UNIX system environment. If you use a file name with the **-f** option, you must include that file name when you are expanding variables [e.g., ${(filename)VARIABLE}].

If a variable name is given without equating it to a value, *set* expects the value to be on *stdin.*

EXAMPLE

Storing a selection made in a menu:

```
   .
   .
   .
name=Selection 2
action=`set -l SELECTION=2`close
   .
   .
   .
```

WARNING

Note that at least one of the allowed switches *must* be used.

Note that a switch must be used for each variable being set/unset.

Note that there is no space between the **-f** option and the filename.

UNIX environment variables (those set using the **-e**) can only be set for the current fmli process and the processes it calls.

When using the **-f** option, unless the file name is unique to the process, other users of the Interpreter on the same machine will be able to expand these variables.

SEE ALSO

env(1), sh(1).

NAME

setcolor – redefine or create a color

SYNOPSIS

setcolor color red_level green_level blue_level

DESCRIPTION

The *setcolor* command takes four arguments; a string naming the color, and three integers defining the intensity of the red, green, and blue components of the color, respectively. If you are redefining an existing color, you must use its current name (default colors are: black, blue, green, cyan, red, magenta, yellow, and white). Intensities must be in the range of 0 to 1000. The function returns the color's name string.

EXAMPLE

```
        `setcolor blue 100 24 300`
```

NAME

shell – run a command using shell

SYNOPSIS

shell command [command] ...

DESCRIPTION

The *shell* command takes each of its arguments and puts them together separated by a space and passes this command to your shell (**$SHELL** if set, otherwise **/bin/sh**).

EXAMPLES

Since the Interpreter does not support background processing, the *shell* built-in could be used instead.

```
`shell "build prog &"`
```

The shell's built-in test can be useful. This will test to see if field2 of a form is a file.

```
valid=`shell test -f $F2`
```

WARNING

The arguments will be concatenated using spaces, which may or may not do what is expected. The variables set in local environments will not be expanded by the shell because "local" means local to the current process.

SEE ALSO

sh(1), test(1).

NAME

vsig – synchronize a co-process with the controlling FMLI object by flushing all output from the co-process into the pipe and sending a signal to FMLI.

SYNOPSIS

vsig

DESCRIPTION

The *vsig* executable sends a SIGUSR2 signal to the controlling FMLI process. This signal/alarm causes FMLI to execute the developer keyword **checkworld**, which causes all posted objects with a "reread" descriptor evaluating to TRUE to be reread.

EXAMPLES

The following is a segment of a shell program:

```
echo "Sending this string to an FMLI process"
vsig
```

The *vsig* command will flush the output buffer *before* it sends the SIGUSR2 signal to make sure the string is actually in the pipe created by the *cocreate* built-in.

SEE ALSO

coproc(1F), signal(2).

I

Index

Index