ELF User's Guide

This manual was produced using Applixware.

Printed: February 2000

Contents

Chapter 4 ELF Language Elements

Chapter 5 The ELF Debugger

Chapter 6 Error Handling

Chapter 7 Dialog Box Editor

Chapter 8 ELF Control Reference

Chapter 10 Advanced Dialog Box Programming

Chapter 11 The Bitmap Editor

Chapter 12 ELF Tasking

Figures

Tables

Preface

About This Manual

The *ELF User's Guide* describes how to use the ELF language and its elements. This book does not explain how to program. It assumes that you can program in a higher level language such as C, Basic, Fortran, or COBOL. This book does explain how to use the ELF programming language to create macros.

Conventions Used in This Manual

The following typeface conventions are used throughout this manual:

Helvetica	Helvetica text indicates that this option or object appears in the document window. For example, "Type the name of the document in the File name entry area." File names and directories are also indicated by Helvetica text. For example, "This file is located in your axhome directory."

	Applixware keys are printed in a Helvetica uppercase typeface. For example, "Press the TAB key."
Helvetica Bold	Bold Helvetica text indicates an option to choose or text to type. It usually appears in numbered steps as shown in the following example: 1. Type **2.5** in the Line spacing entry area. 2. Click **Apply**.
Italics	Words are italicized for emphasis or to draw your attention to a new term. For example, "*Do not* press the RETURN key," or "This action is called *word wrapping*." Italic type is also used to indicate variable information, as in "Put Applixware in the /user/*your_name* directory."

Menu Name → Option Name	Whenever you see a reference to a menu option, the option is identified using the following notation: Menu Name → Option Name For example, "Choose **File** → **Save**."
OK and Apply	When numbered instructions are included in the text, we omit the final "Click **OK** or **Apply**" statement for brevity. When this manual instructs you to CTRL-click or SHIFT-click, press and hold down the CONTROL key or SHIFT key as you click.

Applixware Window Environment and Interface

Consult your window manager and hardware documentation if you need information about how to operate in your window environment.

1 About ELF

This chapter introduces the ELF programming language, and the tools that are available to the ELF programmer. The following topics are discussed:

- Introducing ELF
- ELF Programming Tools
- ELF Programs
- Files and Directories

Introducing ELF

ELF (E*xtended* L*anguage* F*acility*) is a powerful, flexible programming language that is bundled with Applixware. Using ELF, you can write simple macros that interact with your Applixware documents, or you can write complex, robust, stand-alone applications.

ELF includes a library of almost 4000 macros, which allow you to control and manipulate every aspect of an Applixware document. Some macros allow you to interact with the underlying operating system, or mail your document to a co-worker in Tibet.

ELF includes a complete set of development tools to help you develop and run ELF programs. The following tools are discussed:

- The Macro Editor
- The ELF Compiler
- The ELF Debugger
- Dialog Box Editor
- Bitmap Editor
- Applixware Macros

Macro Editor

ELF programs are composed of one or more macros. Macros are contained in macro documents. You create or edit macro documents using the Macro Editor. The Macro Editor includes a compiler and a

debugger. Creating macro documents with the Macro Editor is discussed in Chapter 3, "Macro Editor."

ELF Compiler

ELF is a compiled language. After you create a macro, you must compile it before you run it. You compile macros using a menu option in the Macro Editor.

For a macro to compile successfully, the macro statements must conform to proper ELF syntax. The components of the ELF language and the way in which they must be used are described in Chapter 4, "The ELF Language Elements."

If there are syntax errors, ELF identifies the errors by placing error messages in the macro document near the locations where the errors occur. You can then correct the errors and attempt to compile the macro again.

Even if a macro compiles successfully, that does not necessarily mean it will run successfully. Only simple syntax errors are checked during compilation. Many errors in program logic can be detected only at run time. For example, if you misspell the name of a macro call, the misspelling is not identified during compilation. It is only when ELF attempts to run the macro with the misspelled name that the error is discovered.

If an error is encountered during macro execution, ELF displays an error message dialog box that describes the error and identifies the line number in your macro document where the error occurred.

ELF Debugger

The ELF debugger allows you to find logic problems in your ELF program. Using the ELF debugger, you can single-step through the

execution of your macro one statement at a time, set breakpoints, test variables during program execution, and so on. The ELF debugger is described in detail in Chapter 5, "The ELF Debugger."

Dialog Box Editor

The ELF dialog box editor allows you to design a graphical user interface for your ELF macro. The dialog box editor includes a set of event-driven widgets. By checking for user events, such as a button click or text entered in a field, you can design your program to react to different conditions and user requests. Dialog boxes turn your ELF macro into a fully interactive program.

Dialog box programming is one of the most important parts of ELF programming. A detailed discussion of dialog boxes starts in Chapter 7, "Dialog Box Editor."

Bitmap Editor

The bitmap editor allows you to design icons that appear in the Applixware toolbar. You can connect these icons to ELF macros that you use frequently, making them quickly and easily available to users. Ths Bitmap editor is discussed in Chapter 12, "The Bitmap Editor."

Applixware Macros

Perhaps the most important tool of the ELF programmer is the Applixware macro library, a collection of about 4000 routines that interact with Applixware, and your operating system. Applixware macros can be roughly divided into six groups:

- Spreadsheet macros work with Applix Spreadsheets. These macros have an SS prefix, such as SS_CHART_CREATE@.

  This macro family also includes Applix Real Time. Real Time macros have an RT prefix, such as RT_ENABLE@.

- Words macros work with Applix Words. These macros have a WP prefix, such as WP_BORDER_MARGINS@.

  This macro family also includes the Applix Macro Editor macros, and the Applix HTML macros. The Macro Editor macros have an ME prefix, such as ME_COMPILE@. The HTML have an HTML prefix, such as HTML_OPEN_URL@.

- Graphics macros work with Applix Graphics. These macros have several prefixes, such as GFX (GFX_CHART_DRAW@), GR (GR_GET_PAGE@), and GE (GE_IMPORT_GIF@). GFX macros are specific to the Applix Graphics application, and require that an Applix Graphics window be open. Other Graphics macros work with any Applixware document.

- Mail macros work with Applix Mail and OpenMail. Mail macros have a MAIL prefix, such as MAIL_SEND_MESSAGE@. OpenMail macros have an OM prefix, such as OM_SEND_CURRENT_MESSAGE@.

- Data macros work with Applix Data. These macros have a DBASE prefix, such as DBASE_CONNECTION_RESET@. This macro family includes a number of macros that interact with an SQL database. These macros have an SQL prefix, such as SQL_CONNECT@.

- Base macros are any macros that do not fit into one of the categories described above. These macros perform many functions, such as interact with your operating system (COPY_FILE@), perform arithmetic operations (ABS@), or manipulate data structures in your ELF program (ARRAY_APPEND@).

Applix Builder

ELF is a personal productivity tool, designed to allow you to automate and customize your Applixware desktop.

For large-scale applications development, Applix has designed Builder, an object-oriented, cross-platform programming environment. Builder allows you to rapidly implement powerful applications that port to operating systems across your enterprise.

ELF is used in Builder applications for programming methods. For more information on Builder see the *Applix Builder* manual.

ELF Programs

An ELF program is a macro or collection of macros that perform a particular operation. An example of a very simple ELF macro follows.

This ELF macro accepts the name of a file and a directory location from the user, then copies the file to the specified directory. The ELF statements in this macro are described in Chapter 4, "ELF Language Elements."

ELF Code	Comments
MACRO Backup VAR file, dir	The name of the macro is declared in the MACRO statement.
file = PROMPT@("Name of file to copy") dir = PROMPT@("Directory in which to place copy") IF DIR_EXISTS@(dir) = FALSE CREATE_DIR@(dir)	ELF macros packaged with Applixware end in an at-sign character (@). The PROMPT@ macro waits for keyboard input from the user before continuing.
COPY_FILE@(file, dir) INFO_MESSAGE@("Copy of file "++file++" placed in directory "++dir)	The INFO_MESSAGE@ macro displays a box containing text on the screen.
ENDMACRO	All macros must end with ENDMACRO.

All ELF macros must begin with a MACRO statement and end with an ENDMACRO statement.

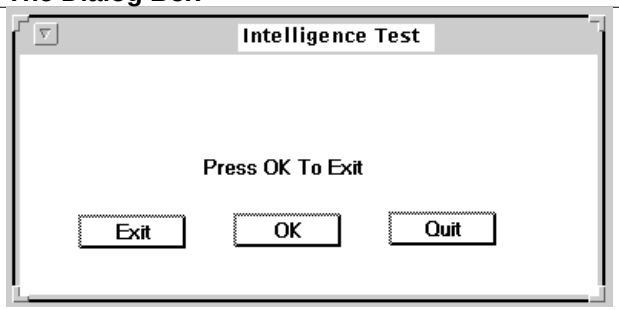
Variables must be declared before they are used. All variables are declared with the VAR statement.

When you run an ELF macro, statements in the macro are executed from top to bottom unless a statement, such as a GOTO, specifically directs program execution to another location.

Dialog Box Programs

The vast majority of serious ELF programs involve a dialog box. All dialog box programs are structured similarly. The example that follows shows a simple dialog box program:

The Dialog Box



The ELF Code

```
macro Example2

var dbox, exit_cond
dbox = DB_LOAD@("genius.d")

while exit_cond <> "OK"

DB_DISPLAY@(dbox)
exit_cond =
DB_EXIT_CTRL@(dbox)

if exit_cond <> "OK" then
  info_message@("You are not
a genius.")

WEND

info_message@("You are a
genius!!")

endmacro
```

This example shows the basic structure of a dialog box program. Those elements are:

- Two declared variables: one for the dialog box (dbox, in this example), and one for the exit condition of the dialog box (exit_cond).
- The ID of the dialog box is assigned to the dbox variable using the DB_LOAD@ macro.
- A WHILE loop displays the dialog box using the DB_DISPLAY@ macro. Control passes to the dialog box, which waits for a user action.

- Each time the user does something, such as click a button, an exit condition is generated and control passes from the dialog box to the macro.
- The WHILE loop tests for exit conditions to determine what the user did, acts on the condition, then re-displays the dialog box.

Dialog box programming is explained in more detail, starting in Chapter 7, "Dialog Box Editor."

Files and Directories

The following files and directories, and system variables are of interest to the ELF programmer:

- login.am
- logout.am
- syslogin.am
- /user/username/axhome/macros
- /install_dir/axlocal
- elf_search_list@

Each of these elements of the ELF programming environment are described in the sections that follow.

The login.am File

A login.am file is useful for customizing Applixware start-up procedures to meet your specific needs. Among the procedures you can perform with a login.am file are:

- Install macro documents that might not otherwise be installed at start-up.
- Display certain Applixware applications or documents when you start up Applixware.
- Include a custom start-up screen that displays a welcome message or provides information to someone using Applixware.

Creating the login.am File

You can automatically install macro documents at start-up by including macros to install the documents in your login.am file. When Applixware starts, it checks for a login.am file and runs the macros contained in the file.

This section describes the procedures for creating a login.am file that contains macros to install macro documents. You can also include other statements in your login.am file to run other procedures at start-up.

A login.am file contains a LOGIN macro that includes macros that are to be run at start-up. When you start Applixware, it checks your /<*home_dir*>/axhome/macros directory for a file named login.am. If the file exists, Applixware runs the LOGIN macro in the file as part of the start-up procedure.

Refer to the section, "Search Paths" later in this chapter for information on the default search paths used by the Macro Editor and how to customize your search path.

To create a login.am file:

1. Start the Macro Editor by clicking on the Macro Editor icon on the Applixware Iconbar.
2. Type the following statement as the first line of the login.am macro:

   macro login

3. Following the macro statement, include the macro calls that you want to run at start-up.
4. Type the following statement as the last line in the document:

   endmacro

   Your login.am macro might look similar to the following:

5. Choose **File → Compile & Save.**

   The macro syntax is verified and the Save As dialog box displays.

6. In the File name entry area, type the name login as the file name and click on **Save**.

Applixware automatically adds the file name extension .am to the file when it is saved.

7. Choose the **File → Exit** option to exit the Macro Editor.

The next time you start Applixware, the macros specified in the login.am file are run. You can edit the login.am file at any time to add or delete macros from the file.

The logout.am File

If you want to run macros when you exit Applixware, you can create a logout.am file in your /user/*username*/axhome/macros directory. If this file contains a macro called logout, Applixware runs that macro when you exit Applixware. The following example logout.am file displays a message each time you exit Applixware:

```
MACRO Logout
        INFO_MESSAGE@("Goodbye "++USERNAME@())
ENDMACRO
```

Refer to the previous section "The login.am File" for information on creating and installing this file.

The syslogin.am File

If you have macros that you want all users to run when they start Applixware, you can configure the syslogin.am file to run those macros. Applixware runs the syslogin macro in the syslogin.am file before the login macro for each user.

NOTE: syslogin.am always runs before login.am.

If the syslogin.am macro file is placed in the /install_dir/axlocal directory, it affects all Applixware users. It can also be placed in /user/*username*/axhome/macros. In this case, it affects only the individual user.

Creating syslogin.am

To create a syslogin.am file:

1. Start the Macro Editor.
2. Type the following statement as the first line of the syslogin.am macro:

   macro syslogin

3. Following the macro statement, include the macro calls that you want to run at start-up.
4. Type the following statement as the last line in the document:

   endmacro

5. Choose **File → Save As.**
6. Save the file as syslogin, in your /install_dir/axlocal directory.

After you install the macro and edit your syslogin.am file, have all users exit from Applixware. When users restart Applixware, the macro runs automatically.

For an example of a syslogin.am macro refer to "Table Edit Macros" in Appendix B of the *Applix Data* manual.

Search Paths

The sequence of directories that Applixware uses to look for a command or file is called the *search path*. The following is the default search path for macros:

1. Your /<*homedir*>/axhome/macros (or equivalent user-specific) directory
2. The /<*install_dir*>/axlocal directory
3. The /<*install_dir*>/axlocal/eng directory
4. The /<*install_dir*>/axdata/eng/Demos directory
5. The /<*install_dir*>/axdata/eng directory
6. The /<*install_dir*>/axdata/elf directory

You can customize your search path by modifying the elf_search_list@ system variable. At startup, the elf_search_list@ system variable contains a list of pathnames that represent the default ELF search path. ELF looks at the directories named in this variable for macros, image files, dialog boxes, object files, and so on. For example:

```
names = "/user/john/macros", "/ax/lpgms", "/ax/pgms/axelf"'
SET_SYSTEM_VAR@("elf_search_list@", names)
```

This statement says that ELF should look in the three named directories for information and in only these three named directories. See "Search Paths" in Chapter 1, "What is Applixware," of the *System Administrator's Guide* for more information.

/user/*username*/axhome/macros

This directory contains your macros. These are automatically installed for you at startup. In addition, the Keystroke Recorder and the Macro Editor use user/*username*/axhome/macros as the default directory when you save a macro. Macros in nested directories, such as user/*username*/axhome/my_macros, are not automatically installed at startup unless you explicitly modify your search path to include them.

The user/*username*/axhome/macros directory is created when you save your first macro.

/*install_dir*/axlocal

The macros in this directory are automatically installed for all users of Applixware. Macros in nested directories, such as /*install_dir*/axlocal/our_macros, are not automatically installed at startup unless you explicitly modify your search path to include them.

Running Macros

To run a macro, press F8 from any place in Applixware. The Run Macro dialog box displays.

Enter the name of the macro to run, and click OK. You can also select a previously-run macro from the list box.

You can run any macro as long as that macro meets one of the following criteria:

- The Macro was in your user/*user_name*/axhome/macros directory when you started Applixware.

- The Macro was in /*install_dir*/axlocal directory when you started Applixware.
- You recorded the macro and saved it in your user/*user_name*/axhome/macros directory.
- You successfully compiled the macro with the Macro Editor. Entering and compiling macros in discussed in detail in Chapter 4, "Compiling and Saving Macros."
- The macro is built into your release of Applixware. There are 4000 macros built into Applixware. For example, the macro WP_APPLICATION_DLG@ starts Applix Words.
- The macro is in another directory that is included in your macro search path.

Running Macros from the UNIX Command Line

You can run macros from the UNIX command line by running Applix Applix -call. The format for applix -call follows:

applix -call *macroname* arg1 arg2 arg3... arg*n*

The macro name must be a macro in the current directory. Pathnames are not supported. The name must have no extension. Do not add the .am extension to the command line. When the macro completes, Applixware exits with the exit code returned by the macro.

When you run a macro with applix -call, several Applixware initialization events are skipped:

- The login and syslogin macros to not execute
- The mail notifer does not turn on
- Applixware does not check for files in back up

2 Recording Macros

There are two ways to create an ELF macro. You can type the ELF code in yourself, or you can record an ELF macro using the Record Macro menu option. This chapter describes creating ELF macros through keystroke recordings. The following topics are described:

- Creating macro documents
- Macro Directories

Creating Macro Documents

The Macro document is a file structure similar to the documents you create in the other Applixware applications—Words, Graphics, Data, and Spreadsheets. All macro documents share the following characteristics:

- They are ASCII files. You can use standard UNIX commands like grep and more with macro documents. You can use the Macro Editor as a replacement for vi or emacs.
- They have a .am extension. You can change this extension through the menus. All .am files shipped with Applixware are macro documents.

A Macro document contains a collection of ELF statements. These statements combine to form one or more macros. A macro performs an action which may be simple or complex depending on your needs.

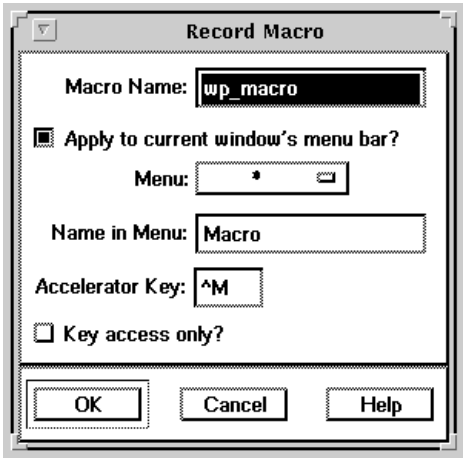
There are two ways to create a Macro document:

- Using the Keystroke Recorder. Whenever you use the Keystroke Recorder, a Macro document that contains macro statements representing the key presses and menu selections is automatically created.
- Using the Macro Editor. You can type the macro or macros into a new Macro document using the options in the Macro Editor application.

Macro Recording Example

To record an ELF macro that inserts a copyright character (©) into an Applix Words document, follow these steps:

1. From Applix Words, Choose ✶ → **Record Macro**. The Applix Words cursor should change to indicate that keystrokes are being recorded.
2. Select **Insert** → **Special Characters**.
3. Click the copyright character.
4. Press Enter.
5. Choose ✶ → **Record Macro**. The Applix Words cursor should change to indicate that keystrokes are no longer being recorded.
6. Choose ✶ → **Record Macro**. This dialog box appears:



7. Press **OK**. A file named wp_macro.am is written to your axhome/macros directory.

If you look at wp_macro.am with any text editor, the contents looks like this:

```
macro wp_macro
'
' Variables used:
'
        var CODE
        var FONT_NAME
'
' Macro calls:
'
        CODE = "©"
        FONT_NAME = "Palatino"
        WP_ENTER_CHARS_FACE@(CODE,FONT_NAME)
endmacro
```

This is a fully functional ELF macro, which you can modify, compile and run.

Macro Directories

There are two places that macros are stored:

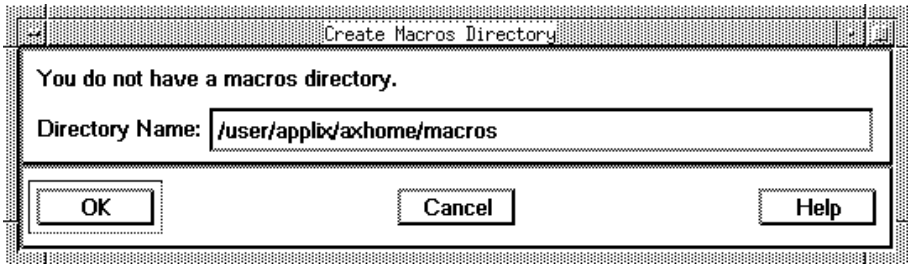
- /<*homedir*>/axhome/macros where *username* is your login name. The macros in this directory are available only to you. All keystroke Macro documents are saved in this directory. Each Applixware user has a separate macros directory for locally stored and used macros.
- /<*install_dir*>/axlocal where *install_dir* is the directory in which Applixware is installed on your system. All users of Applixware can run these macros.

Normally, keystroke macros are saved in your macros directory. If you want macros to be placed in the local macros directory, you must move or copy them there yourself.

Macros that are stored in either of these two directories are compiled and installed for you when you start Applixware.

Creating Your Macros Directory

The first time you attempt to store a macro made from a keystroke recording or the first time you attempt to save a Macros document in the Macro Editor, Applixware prompts you to create your macros directory.



You can change your macros directory by entering a new directory in the Directory Name entry field.

Once your macros directory is created, all keystroke Macro documents are automatically placed in this directory when they are saved. Macros saved in this directory are installed automatically when you start Applixware.

Changing the Macros Directory

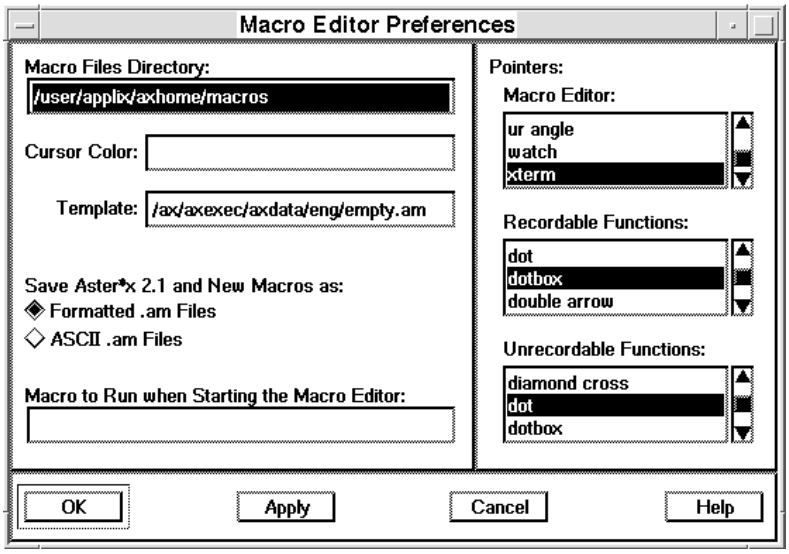
You can change the directory used as your Macros directory even after it has been established. If you change your Macros directory, you

should move all your macros to the new directory so they can be located by Applixware.

To change your Macros directory:

1. Choose ✶ → **Macro Editor** from any application or click on the Macro Editor icon on the Applixware Iconbar.

2. Choose ✶ → **Macro Editor Preferences** to display the Macro Editor Preferences dialog box.



3. Type the full pathname of the directory you want to use as your macros directory in the Macro Files Directory entry area.

   The default is /<*home_dir*>/axhome/macros, where *username* is your user name.

3 Macro Editor

This chapter describes the Macro Editor. The Macro Editor is an ASCII editor that lets you edit, change, compile, debug, and save ELF macros and Macro documents.

Using the Macro Editor

The Macro Editor is based on the functionality of Applix Words, but with a more compact user interface. Options that have similar functionality in Words and Macros are described briefly in this chapter. Refer to the *Applix Words* manual if you need a more thorough explanation of any of these options.

The Macro Editor is designed for application developers, and lacks many of the powerful word processing features of Words. You can customize your version of the Macro Editor using the ✶ → Customize Menu Bar. If you want to add certain features from the Words menu to the Macro Editor's menu, just add the menu name and its corresponding macro from the Words menu.

You can use the Macro Editor to create or edit macro documents. The Macro Editor includes a compiler that checks your macro for syntax errors, a debugger for debugging macros as you create them, a Dialog Box Editor that allows you to create and modify dialog boxes, and a Bitmap Editor that lets you edit bitmap images.

Macro Editor Preferences

You can determine the style of the mouse pointer, the name of your macros directory and your default save format using the ✶ → Macro Editor Preferences.

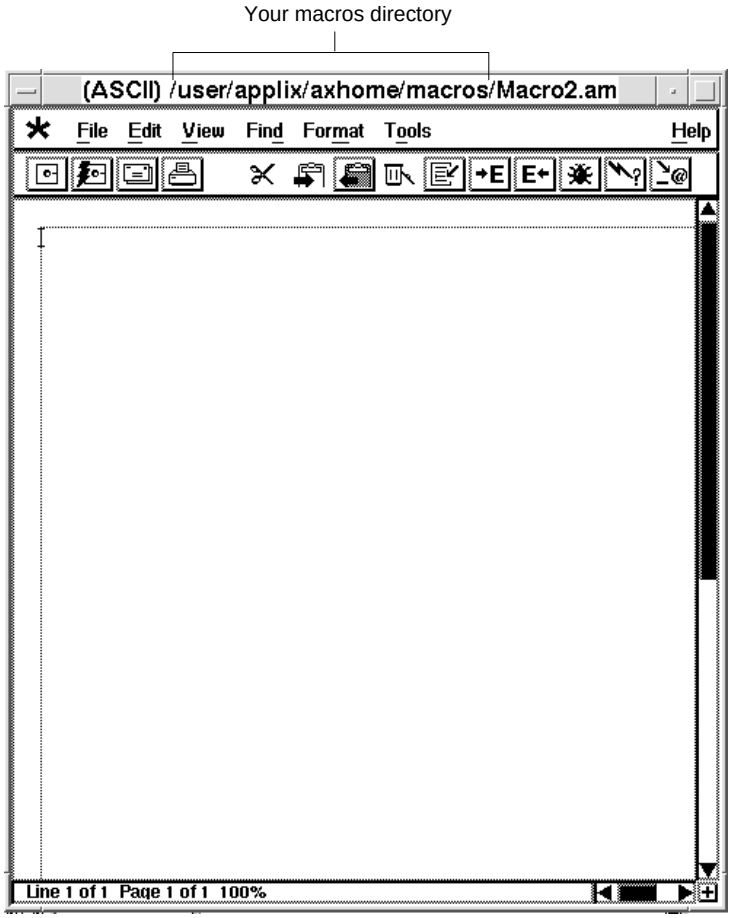
Changing Macro Editor Defaults

By default, the Macro Editor uses 12-point times-roman font. To change this default font, follow these steps:

1. Locate the file empty.am in the *install_dir*/axdata/*lang*, where lang is the language-specific directory.
2. Copy empty.am to empty.aw.
3. Open empty.aw with Applix Words.
4. Make the appropriate changes to the file. You can change the font or the font size, or make any other changes you want.
5. Save empty.aw.
6. Copy empty.aw to empty.am.

Creating a Macro Document

To create a macro document using the Macro Editor, choose ✶ → **Macro Editor** from any Applixware application or click on the Macro Editor icon on the Applixware Iconbar.



A new, blank macro document appears in a Macro Editor window. Notice that the Macro default document automatically opens into your macros directory. Any macro you type in and save here is automatically installed into that directory.

To create a macro document when you first start Applixware, type applix -me at your operating system prompt.

Opening a Macro Document

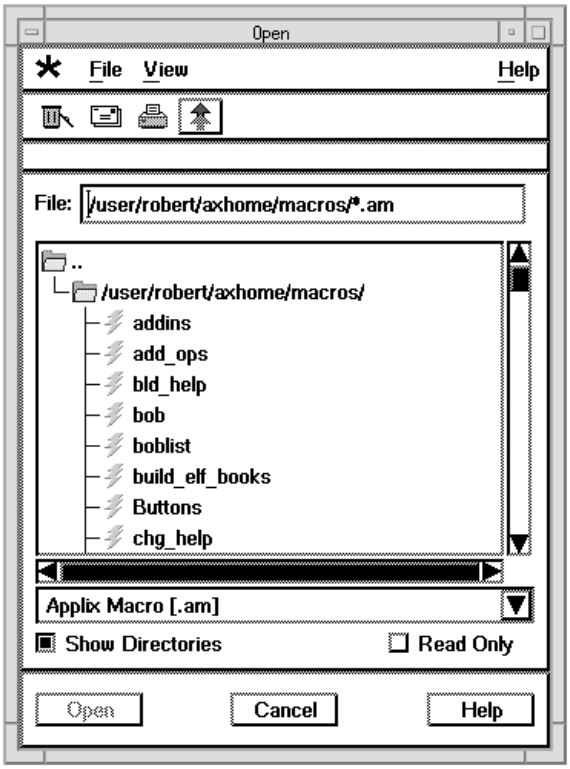
Applixware provides a variety of ways to open an existing macro document. You can:

- Choose a macro document from the Directory Displayer. For information about the Directory Displayer, see "The Directory Displayer" in the *Getting Started* manual.
- Open a macro document from a Macro Editor window.
- Open a macro document when you start Applixware.

To open a Macro document from a Macro Editor window:

1. Choose **File → Open** to display the Open dialog box.

   The Open dialog box displays the contents of your macros directory.



2. Double-click on the macro document to open it, or click the name of the document you want to open in the list box, then click on **Open**.

To open a macro document when you start Applixware, type applix *name*.am at your operating system prompt, where *name* is the full pathname of the macro document you want to open.

Viewing Options

The Macro editor offers various viewing options:

View → ExpressLine

Turn on to display a bar of buttons for frequently used operations.

View → Ruler — Turn on to display the graphical ruler.

View → Boundaries

Turn on to display all formatting boundaries in the document such as headers, footers and margins.

View → Delete Error Messages

Turn on to remove the error messages inserted into this macro document as a result of compiling.

View → Zoom — Choose one of the magnification size options to display the document onscreen in a variety of magnification sizes, larger or smaller. Sizes less than 100% will shrink the display of the text; sizes greater than 100% magnify the text. 100% displays the document at full size. The Zoom size does not affect the printed document. The document always prints at 100% size.

Inserting Text

Once you open a Macro document, you can type text into it just as you type text in a Words document. Text automatically word-wraps. For more information, see "Typing Text" in Chapter 2, "Getting Started With Words" in the *Applix Words* manual.

Inserting Special Characters

You can include special characters in your macro document. Choose Format → Special Characters and select the group and character you want to insert.

NOTE: If you save your macro document in ASCII format, all formatting information is lost. Text appears in the default mono-spaced font for your terminal. Margin settings are also lost.

Character and Paragraph Settings

You can set the typeface, size, bold, and italic text attributes in macro documents using the Format → Character Settings option. Select the text or line to change and choose the appropriate typestyle from this dialog box. If nothing is selected when you change a setting, the default attributes are changed throughout the document.

You can set the paragraph spacing, indent levels, hyphenation and justification using the Format → Paragraph Settings option.

NOTE: If you save your macro document in ASCII format, all paragraph and character settings will be lost.

Promoting and Demoting Text

The Macro Editor offers the ability to promote and demote lines of text using Format → Promote and Format → Demote. This is useful for differentiating nesting levels when writing a macro document.

NOTE: If you save your macro document in ASCII format, indent level information is lost.

See "Indenting and Aligning Paragraphs" in the *Applix Words* manual for more information.

Shift Case

To change the case of selected text, choose Edit → Shift Case. Choose this option again if needed to change to the case you want.

The first word in the selected text determines the initial case setting. The case changes in the following order:

UPPER CASE
Initial Capitals
lower case

If you select UPPER CASE text and then choose Edit → Shift Case, the text changes to Initial Capitals.

If you select text in Initial Capitals, then choose Edit → Shift Case the text changes to lower case.

If you select lower-case text, then choose Edit → Shift Case, the text changes to UPPER CASE, and so on.

Mixed case, such as ApPLiX, changes to lower case (applix). The original mixed case cannot be retrieved using Edit → Shift Case. Choose Edit → Undo to return to mixed case.

Completing Macro Syntax

The Macro Editor includes a facility to help you enter the name and arguments of the ELF built-in macros. If you enter the name of a macro in your macro document, but you cannot remember the arguments, you can place the cursor inside the text of the name, and press the *Express*line icon that looks like this:

The Macro Editor completes the indicated macro with the name and arguments of the macro that most closely matches the text you typed in.

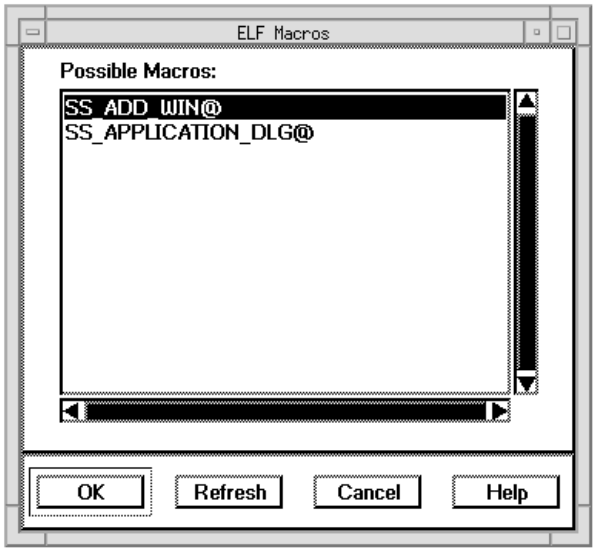
For example, suppose you type in the following text in the Macro Editor:

```
macro test
    SS_APPLICATION_DLG@
endmacro
```

To get a complete list of arguments for the macro, press the macro completion icon in the *express*line. The Macro Editor completes the text you typed in, as shown here:

```
macro test
SS_APPLICATION_DLG@([menubarID][, windowless][, hookless])
endmacro
```

If you enter an incomplete name, such as SS_A, and press the macro completion icon, the Macro Editor displays a list of macros that are closest to what you typed in. That list looks like this:



Select the macro that you want, and click **OK**. The macro name and arguments are written into the macro document.

Cutting, Copying and Pasting Text

Choose **Edit → Cut** to remove selected text.

Choose **Edit → Copy** to make a copy of the selected text to be pasted elsewhere in this or another document.

When you choose Edit → Cut or Edit → Copy, the selected text is placed in the clipboard. In the case of an Edit → Cut the text is removed from the macro document.

Once material is on the clipboard, you can use Edit → Paste to move the cut material or to insert the copied material into the macro document at the current cursor position.

NOTE: The contents of the clipboard are overwritten each time you use Edit → Cut or Edit → Copy.

Selecting All Text in a Macro Document

Choose **Edit → Select All** to select all text in the Macro document.

Deleting Selected Text

Choose **Edit → Delete** to delete all currently selected text.

Unlike Edit → Cut, Edit → Delete does not save the deleted text in the clipboard. You cannot retrieve deleted text unless you use Undo.

Undoing Changes

Choose **Edit → Undo** or press the UNDO key (F2) to restore your macro document to its condition prior to the last edit performed.

An edit is considered any operation that alters the contents of the Macro document. For example, if you position the cursor at an insertion point and type a few characters, but then decide you want to delete the characters, you can choose Edit → Undo to restore the line. The Undo menu option changes dynamically to reflect the last action

you performed, such as Undo Cut. This is the action that will be undone.

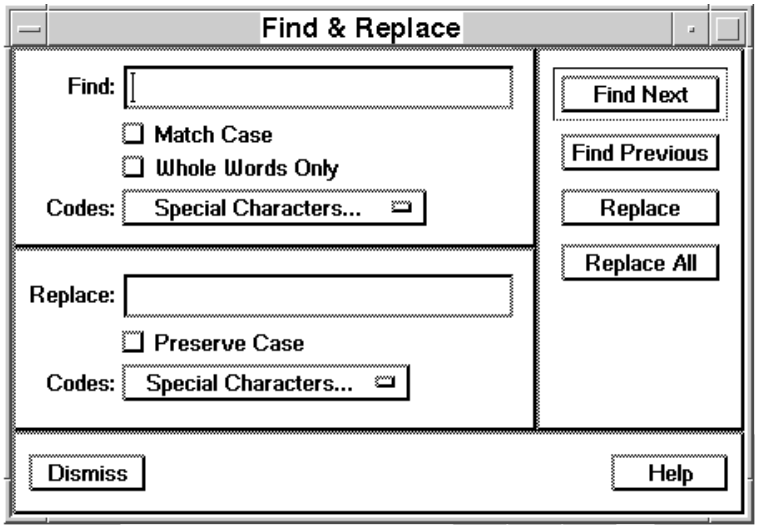
You can use Undo repeatedly to walk back through your actions up to the last File → Save operation. Redo restores any action you "undid."

You can use Edit → Redo to redo an action you have just reversed using Edit → Undo. For example, if you delete material, then choose Edit → Undo to restore it, you can choose Edit → Redo to delete it again.

Searching for Text

To find and select text in a document:

1. Choose **Find → Find & Replace** to display the Find & Replace dialog box.



2. Type the text you want to find in the Find entry area.

   You can enter 1 to 80 characters including numbers, letters, symbols, blank spaces, and wildcards.

   Click on one of the Codes: options to search for a special character, the clipboard content, a word start or end, or a paragraph start or end.

3. Click on **Match Case** to find only text typed in the same case as the text you entered in the Find entry area.

   Click on **Whole Words Only** to find only whole words including trailing spaces or punctuation marks.

4. Click on **Find Next** to initiate the search in a forward direction. Macros searches from the text cursor to the end of the document. After it reaches the end of the document, it then searches through headers, footers and footnotes.

5. Click on **Find Previous** to initiate the search in a backward direction. Macros searches from the text cursor to the beginning of the document.

Replacing Text

Choose **Find → Find & Replace** to substitute one text pattern with another:

1. Complete the upper portion of the dialog box−the Find fields−just as you would for searching as explained in the previous section.

1. Type the text to substitute in the Replace entry area.

2. Turn on **Preserve Case** to enforce the case in the Find box on the replaced text.

3. Click on one of the Codes options to replace with a special character, macro, or the clipboard content.

After typing the search information, determine if you want to replace just the currently selected text or all occurrences of the search text.

Replace	Click here to replace just the current selection.
Replace All	Click here to replace all occurrences of the selected text.
Find Next	Click here to move forward to the next occurrence.
Find Previous	Click here to move backwards to the previous occurrence.
Cancel	Click here to ignore the find and replace information and return to the document.

You can also use the Find → Next and Find → Previous menu options to find subsequent or prior occurrences of a word.

Moving Around in Macro Documents

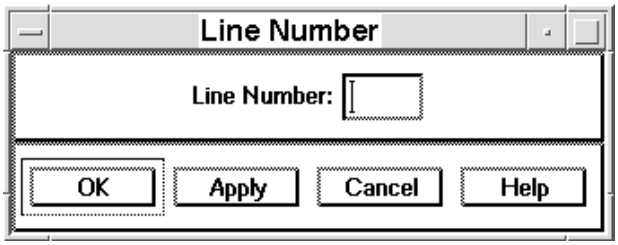
Use other Find menu options to move the cursor to specific positions within the macro document. Use Find → Page to go to a specific page number.

Moving to a Specific Line Number

The term *line* in this manual means all text in front of a paragraph marker.

To move the cursor to a specific line (paragraph) in a document:

1. Choose **Find → Line**.



2. Type the number of the line to which you want to move the cursor. Lines are numbered from 1.

Mailing a Macro Document

If you are licensed for Applix Mail, you can send your macro document across a network using File → Send. You can send your document as a Macro document, or you can export the document to file formats compatible with other software.

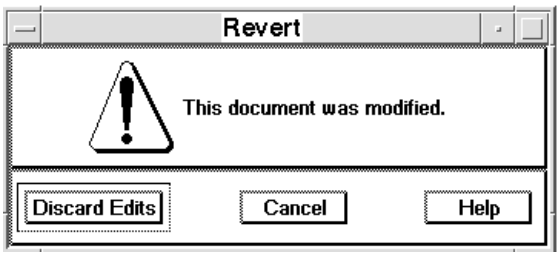
Reverting a Macro Document

Choose **File → Revert** to eliminate all changes made to a document since the last time it was saved.

This option is a useful way to quickly eliminate document changes if you find that you do not want to save any of the changes you have made.

To revert a document to its last saved state:

1. Choose **File → Revert**.



2. To revert the document, click **Discard Edits**.

To cancel the revert, click **Cancel**.

Saving a Macro Document

You save a macro document in the Macro Editor the same way you save documents in Words, Graphics, Data, and Spreadsheets. Saving has special implications when the macros themselves have been edited. See "Compiling and Saving" later in this chapter for details about re-compiling and saving edited macros and Macro documents.

Note that if you save a macro document in ASCII format, no formatting or special characters will be saved. If you subsequently revert, all visible formatting will be lost.

Macro File Formats

There are several file formats in which you can save Macro documents. The default format is set through ✶ → Macro Editor Preferences. You can also specify these preferences by selecting File → Save As from the Macro Editor menu.

Formatted.am	Choose Formatted.am Files to save your macro document in Words (.aw) format. This format allows you to use multiple fonts, include insets, special characters, and use Word macros to manipulate data. This is the default.
ASCII.am	Choose ASCII.am Files to save your macro documents in ASCII format. If you prefer this format, set your ✶ → Macro Editor Preferences format to ASCII.

Compiled format

You can also choose to store either your ASCII or formatted macro documents in a compiled format by turning on the Compiled Format option on the File → Save As dialog box. Macro documents stored in compiled format do not have to be compiled before they are used; however, they may take up more space. Compiled files are machine-independent. Compiled files cannot be opened or edited by the Macro Editor.

Macros stored in compiled format have a .elo extension.

Page Layout

Like other Applixware documents, you can change the margins or print setup on Macro Editor documents using Format → Page Setup. You can print in portrait or landscape orientation. Also, you can have headers and footers in macro documents using Format → Header & Footer.

Refer to "Defining Headers and Footers" and "Page Setup" in the *Applix Words* manual for more information.

Printing a Macro Document

You print a macro document just as you print any Applix Words document.

Deleting a Macro Document

To delete an open macro document:

1. Choose **File → Delete** to display the Delete dialog box.
2. Click **Delete**.

   If you change your mind and do not want to delete the document, click **Cancel**.

NOTE: The document is not recoverable once you delete it.

You can also delete any type of document using the Directory Displayer.

Compiling Macros

Before you can use a macro you must first compile it. If your macro contains syntax errors, the compiler inserts error messages at or near the locations of the errors. Refer to Chapter 11 "ELF Language Elements" for more information on the syntax of your macro statements.

If your macro compiles without error, but does not run as expected, your macro may have a logic error. In this case, you can use the ELF debugger to determine where you code is failing.

See Chapter 5 "The ELF Debugger" for more information on the ELF debugger.

Compiling and Saving a Macro Document

Whenever you make changes to a macro document, you should compile the macro before you leave the Macro Editor. Choosing File → Compile & Save is a convenient way to perform the operations of checking macro document syntax, and saving the document.

If syntax errors are found, the macro document is still saved. In that case, correct the errors and then compile the document again. Refer to Chapter 4 "ELF Language Elements" for more information on the syntax of your macro statements.

Moving to the Next or Previous Error

After you have written a new macro or edited an existing one, you must use File → Compile & Save or File → Compile to check the syntax of your macros. If you encounter errors, they are flagged within the macro document at the appropriate location. See "Compiling and Saving" earlier in this chapter for more information on saving and compiling macros.

Choose Find → Next Error to move the cursor to the next error message in a macro document.

Use Find → Previous Error to move backward to the previous error message.

To remove the error messages from your macro document choose View → Delete Error Messages.

When no more errors are found with either Find → Next Error or Find → Previous Error, a message displays. See Chapter 5 "The ELF Debugger" for more information on the ELF debugger.

Deleting Syntax Error Messages

To remove all syntax error messages generated as a result of choosing File → Compile & Save or File → Compile, choose View → Delete Error Messages.

Error messages are also automatically removed when you print a document.

Installing a Macro Document

Before a macro can be used, it must be compiled and installed in memory. In most cases, macro installation is automatically performed by Applixware. Installation of macros occurs in the following instances:

- When a keystroke recording is made, the macro created by the recording is automatically installed for the duration of the current Applixware session.
- When a macro is called, Applixware automatically runs the macro if it is already installed in memory. If it is not, Applixware searches the directories in your search path for a .elo or .am file matching the macro name. (It looks directory by directory first for .elo files and then .am files.) If it finds a .elo file, it installs the file and runs the macro. If it finds a .am file, it compiles the file, installs all the macros and runs the called macro.

Refer to the section "Search Paths" in Chapter 1 for information on search paths.

NOTE: Applixware looks for a macro by searching for a macro document with the same name as the macro, and then installs it. If a macro does not have the same name as the macro document in which it resides, Applixware cannot find it to install it.

If a macro meets any of the above criteria, it is automatically installed and you do not have to install it explicitly. However, you must install a macro in the following circumstances:

- The name of the macro is different from the name of the macro document in which it resides. For example, if you have a macro called Credit_Report in the file Financials.am, the only way to install the Credit_Report macro is to open Financials.am with the Macro Editor, and compile the file.
- The macro does not reside in any of the search path directories.

If you attempt to run a macro that has not been installed, Applixware displays an error message.

Manually Installing Macros

To manually install a macro for the current session, you have the following options:

- Choose Tools → Install Macro File to install the .am or .elo file containing the desired macro. Or,
- Use INSTALL_FILE@(filename) to install the .am or .elo file containing the desired macro.

Or, for .am files only,

- Open the file in the Macro Editor and compile it.

The easiest way to avoid having to explicitly install macro documents is to place all macro documents in your macros directory and have each document contain a single macro that has the same name as the

document. However, if you have a large number of macros, you might find it convenient to include a collection of the macros you use the most in a single file and install the document when you start Applixware. You can automatically install a document at start-up using a login.am file as described in Chapter 1 "About ELF".

Duplicate Names among Installed Macros

Macro documents can contain more than one macro. While you should avoid using duplicate names for macro documents, you may use the same name for a *macro* that resides in several different documents.

Having multiple macros with the same name can pose a problem if the functionality is different among two or more of the macros. When two macros having the same name are installed in Applixware, the macro that was installed *last* takes precedence. For example:

1. You compile a macro called Test at 1:00 on Monday.
2. You Write a new macro called Test and compile it at 1:30 on Monday.
3. When you press F8 and run Test at 1:35, the macro you compiled at 1:30 is the one that runs.

The easiest way to avoid duplicate macro name conflicts is to use unique names when the contents of two macros differ. If you do use duplicate names, be careful about the order in which macro documents are installed and macros are called. If necessary, re-install a macro document using the Tools → Install Macro File option so that you can use the macro once again.

Exiting the Macro Editor

To exit the Macro Editor, choose File → Exit.

4 ELF Language Elements

This chapter describes the structure of the ELF language. All topics are arranged in alphabetical order. These topics are divided into four groups, as follows:

Macro Constructs - comments, constants, DEFINE statement, ENDMACRO statement, EXTERN statement, FUNCTION statement, INCLUDE statement, MACRO statement, UIMACRO statement

Variables and Data - ARRAYOF, arrays, data types, FORMAT, global variables, local variables, string variables, system variables, VAR statement

Operators - AND, arithmetic operators, EQV, IMP, logical operators, NOT, OR, relational operators, XOR

Program Control - BREAK statement, Case statement, conditional statements, For loops, GOTO statement, IF statement, ON ERROR statement, ON GOTO statement, RETURN statement, WEND statement, WHILE loops

AND Operator

AND is both a logical operator and a bitwise operator.

As a logical operator, AND is used primarily in conditional expressions. For an expression containing an AND operator to evaluate to TRUE, both of the objects in the expression must be TRUE.

AND can also be used for bitwise operations. For example, the following statement displays the number 4 in an information box.

```
info_message@(5 AND 6)  ' 0101 AND 0110 = 0100
```

The following example is an IF statement that tests an expression, and executes two statements if the tested expression is TRUE.

```
IF Buy_Applix_Stock AND Stock_Splits
{
Bank_Account = Bank_Account + 1000000
Goto EARLY_RETIREMENT
}
```

Refer to the section "OR Operator" for related information.

Arithmetic Operators

An arithmetic operator is the part of an expression that specifies the type of operation or calculation to be performed. The expression Budget + .25 instructs ELF to add the numeric constant .25 and the

value of the variable Budget. The addition sign (+) is an example of an ELF arithmetic operator.

ELF supports arithmetic, relational, and logical operators. The following tables list all ELF arithmetic operators. The column "Leading/Trailing Spaces" indicates whether you are required to include spaces before and after the operator when you type the operator.

Table 4-1 Arithmetic Operators

Arithmetic operator	Operation	Example	Leading/Trailing Spaces
-	Negative	-450	optional
^	Exponentiation	2^5	optional
*	Multiplication	50*2	optional
/	Division	100/2	optional
\I	Integer division	100\2	optional
MOD	Remainder operator	22 MOD 4	required
+	Addition	5+5	optional
-	Subtraction	10-2	optional
++	String concatenation	Area code++ Phone	option

ARRAYOF Statement

To declare a variable to be an array of FORMAT variables, you must use the ARRAYOF statement. The format for the ARRAYOF statement is as follows:

```
VAR FORMAT ARRAYOF Formatname Variablename
```

The following code sample uses a FORMAT variable called budget_info. The variable Years is established as a formatted array containing the fields in the budget_info format.

```
FORMAT budget_info
        first_qtr_profit,
        second_qtr_profit,
        third_qtr_profit,
        fourth_qtr_profit,
        year_profit
/*
*       Formats are declared before the start of the macro.
*/
macro budgets

var x
var Format Arrayof budget_info Years

/*
*       This declaration declares the variable Years to be
*       a formatted array containing the fields shown in the
*       budget_info format.
*/

for x = 0 to 5
        Years[x].first_qtr_profit = 1000
next x

/*
*       You can use dot notation to access elements for the
*       formatted array.
*/

        info_message@(Years[4].first_qtr_profit)
endmacro
```

Arrays

You can use an ELF array to store a collection of data. Arrays in ELF are *heterogenous*, which means that the data in a single array can contain any combination of data types. An array is not limited to data of only one type. The following code fragment defines an array with three different types of data:

```
macro heterogenous_array
var x
x = "hello", 1, NULL        ' The array contains three elements
endmacro
```

Declaring an Array

In ELF, an array is declared using a VAR statement. You do not declare the size of an ELF array. The size of the array is maintained automatically by ELF according to the number of elements that you assign to the array. For example, in the following macro, three names are assigned:

```
macro Bears
var bears
bears = "Mama", "Papa", "Baby"       ' bears contains three elements
endmacro
```

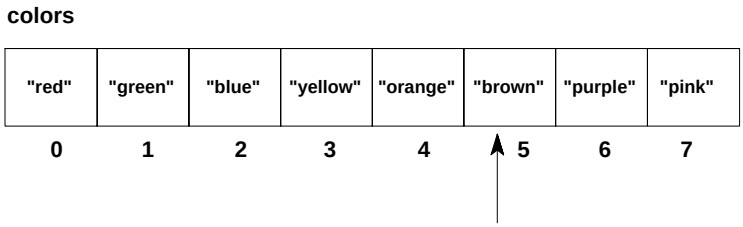
The name of the array must be unique within the ELF macro.

Referencing Array Elements

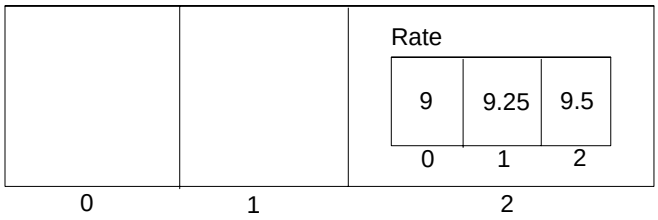
To reference the individual elements of an array, use the following notation:

```
array_name[element_number]
```

array_name is the name you give the array. element_number indicates the location of the element within the array. ELF arrays are zero-based, meaning the first array position is location 0, the second array position is location 1, and so on. The following shows the notation you use to reference the sixth element in the array named colors.



You can include an array as the element of an array. Arrays referenced by other arrays are called *sub-arrays*. In this sub-array example, the Budget96[2] array element contains the sub-array Rate.



You can reference a value contained in a sub-array directly using the element number that corresponds to the value's position in the sub-array. The format for referencing sub-array elements is:

array_name[element_number, sub_array_element,
sub_array_element...]

For example, Budget96[2,0] references the value 9, Budget96[2,1] references the value 9.25, and Budget96[2,2] references the value 9.5 as shown in the following figure.

Arrays can have as many elements as you need. Any element can also contain an array. In this way, you can easily create large multi-dimensioned arrays.

Assigning Values to Arrays

You use the following statement to assign a value to an array element:

```
array_name[element_number]=expression
```

array_name is the name of the array variable. element_number is the position within an array that the value will occupy. The expression is the value to be assigned to the element.

You can assign string values, number values, or other arrays to an array element. For example, the following assignment statements create an ELF array, Name, containing six string values:

```
Name[0]="Smith"
Name[1]="Jones"
Name[2]="MacIntosh"
Name[3]="McCabe"
```

```
Name[4]="Stone"
Name[5]="Vito"
```

Short Form Array Assignments

You can also create the same array using the following short form:

```
array_name=expression, expression, expression...
```

For example, the Name array could be assigned as follows:

```
Name="Smith", "Jones", "MacIntosh", "McCabe", "Stone", "Vito"
```

In this short form array assignment, array values are separated by commas. A value's position within the assignment statement determines its position within an array.

You can assign values to sub-arrays in either of two ways:

- Assign values to the sub-array and then assign the sub-array to the array. For example, if Rate is a sub-array contained in the array named Budget96, you could assign values as follows:

  ```
  Rate = 9, 9.25, 9.5
  Budget96[2] = Rate
  ```

- Assign values to sub-arrays elements directly by referencing the element in your assignment statement. For example, if Rate is a sub-array contained in the array named Budget96, you could assign values as follows:

  ```
  Budget96[2,0] = 9
  Budget96[2,1] = 9.25
  Budget96[2,2] = 9.5
  ```

Short Form Array Assignments Using Braces

ELF also allows you to indicate an array element using braces. For example, the last Name declaration could also be written as follows:

```
Name={"Smith", "Jones", "MacIntosh", "McCabe", "Stone", "Vito"}
```

In this case, the braces add no value. Here are a few examples that show where they should be used.

- When passing variables to a function, braces are easier than packing information into a temporary value. For example, the following two macros perform the same action:

  ```
  macro TEST1(arg1, arg2)
  var packing
  packing = arg1, arg2
  DO_SOMETHING(packing)
  endmacro
  macro TEST2(arg1, arg2)
      DO_SOMETHING( {arg1, arg2} )
  endmacro
  ```

- Braces let you indicate multi-dimensional data. For example:

  ```
  data = {"The", { "quick", "brown"}, "fox"}
  ```

  This statement creates a three-element array. The second element is also an array containing two elements.

  In the following example, a variable with one array element is created:

  ```
  data = {1}
  ```

Although data has only element, ELF interprets it as an array.

In some cases, you need to remove the data from an array but need to keep the variable's type as an array. You can do this in one of two ways:

```
data = TRUNC_ARRAY@(data, 0)
data = {}
```

Refer to "FORMAT Statement," later in this chapter, for more information.

Bitwise Operators

An operator is the part of an expression that specifies the type of operation or calculation to be performed. Bitwise operators are used to create a numeric value from two other numeric values.

The following is an example of bitwise operations in ELF:

```
if X and Y then statement
```

where X and Y are numbers. The corresponding bit of each number are compared. If both corresponding bits are 1, then the solution contains a 1 in that location. For example:

```
            0 0 0 0 0 1 1 0    ' The number 6 in binary
AND         0 0 0 0 0 1 0 1    ' The number 5 in binary
-------------------------------------------------------------------------
              0 0 0 0 0 1 0 0    ' The number 4 in binary
```

Therefore,

```
5 AND 6 = 4          ' Bitwise AND
```

ELF supports arithmetic, relational, and logical operators. The following table lists all the ELF logical operators. The column "Leading/Trailing Spaces" indicates whether you are required to include spaces before and after the operator.

Table 4-2 Bitwise Operators

Logical operator	Operation	Example	Leading/Trailing Spaces
Not	Inverse	NOT 01 = 10	Required
AND	And	11 AND 10 = 10	Required
OR	Or	10 OR 00 = 10	Required
XOR	Exclusive or	10 XOR 11 = 01	Required
EQV	Equivalence	11 EQV 01 = 01	Required
IMP	Implication	01 IMP 00 = 10	Required

Refer to the section "Operator Precedence" for related information.

BREAK Statement

The BREAK statement tells ELF to stop processing the loop and resume execution with the statement following the FOR loop. The syntax for this statement is:

```
BREAK var
```

where var is the counter variable of the FOR loop. var is optional. If it is omitted, BREAK stops processing for the current loop. The following example shows a simple BREAK statement, with no argument.

```
FOR i = 0 to 99
        ...
        if COND = -1
                BREAK           'The Loop is terminated
        ...
NEXT i
/*
*       Execution Resumes Here
*/
```

The following example shows two loops. When the inside loop reaches 50, the BREAK statement terminates both loops.

```
FOR outside = 0 to 99
        FOR inside = 1 to 99
                IF inside = 50
                        BREAK outside
        NEXT inside
NEXT outside
/*
*       Execution resumes here. Note that a simple BREAK
*       statement would have resumed with the NEXT outside
*       statement. By specifying outside in the BREAK statement,
*       both loops are terminated.
*/
```

CASE Statement

The CASE statement is a conditional statement that tests an expression against a set of possible values for that expression. A CASE statement is formatted as follows:

```
CASE OF expression
CASE value1 [, value2]...
    statement
```

```
    statement
    ...
CASE value3 [, value4]...
    statement
    statement
.
.
.
[ DEFAULT
    statement
    ... ]
ENDCASE
```

The expressions in the CASE statement, like all ELF expressions, can either be numeric or string. However, in one CASE statement all expressions must either be numeric or string.

The DEFAULT keyword and the statements that follow it are optional.

```
CASE OF result          ' Result is a string in this example
CASE "S", "T"
    menu_demo1
CASE "F"
    menu_demo2
CASE "V"
    menu_demo3
DEFAULT
    menu_demo4          ' If result does not meet any of the
                        ' previously-specified values.
ENDCASE
```

All statements following the CASE statement are executed until one of the following is encountered:

- Another CASE statement
- A DEFAULT statement
- An ENDCASE statement

After encountering one of these statements, ELF resumes execution with the statement that immediately follows the ENDCASE statement.

Comments

A comment is text in your macro document that is ignored when the document is compiled and executed. ELF supports two methods for placing comments in your macro documents:

- You can comment text on a single line by placing a single quotation mark (apostrophe) before the comment text as shown:

```
MACRO PrintSales  'This macro prints on the Sales
                          ' Dept's  laser printer.
VAR copies
copies = PROMPT@("How many copies?")
...
```

  Any text that follows the quotation mark on the same line is considered a comment and is ignored.

- You can comment large blocks of text on multiple lines using the /* and */ comment symbols. With this method, the symbol /* signals the start of a comment and the symbol */ signals the end of the comment. All text between the two symbols is recognized as a comment. For example:

```
MACRO LoadASCII
 /*
```

```
* This macro opens an operating system file, reads it, and
* writes the contents to a Words document.
*/
    VAR op_file, name
    ...
```

Conditional Statements

ELF provides two methods that allow you to execute groups of statements based on a condition: the CASE statement and the IF statement. These statements are instrumental in the construction of dialog box macros, since you must use conditional statements to determine what action to take after an exit condition has been encountered. See Chapter 8, "Dialog Box Programming", for more information.

Refer to the "CASE Statement" and "IF Statement" sections in this chapter for related information.

Constants

Constants are numerical or string values that do not change throughout an ELF program. ELF provides three constants: TRUE, FALSE, and NULL.

The constant TRUE has the value -1. The constant FALSE has the value 0. The constant NULL is the null datum value.

TRUE and FALSE can be used interchangeably with -1 and 0 in any of your macros or functions. For instance, the following statement tests whether a toggle button is toggled on (value = -1).

```
IF DB_CTRL_GET_VALUE@(dbox,"Toggle1") = TRUE
```

When the return value of a macro is TRUE or FALSE, the macro returns -1 (TRUE) or 0 (FALSE). Therefore, if you have a statement that displays the return value of such a macro, the value -1 or 0 is displayed. For example, suppose your macro includes the following statements for returning the value of a toggle button in a dialog box:

```
VAR value
value = DB_CTRL_GET_VALUE@(dbox, "Toggle1")
INFO_MESSAGE@("The value of toggle 1 is" ++value)
```

If value is determined to be TRUE, the INFO_MESSAGE@ macro displays the following in a dialog box:

```
The value of toggle 1 is -1
```

User-Defined Constants

To assign a name to a constant value, use the DEFINE statement. After you assign a name to a constant, you can use the name in place of the constant. This can enhance the readability of your macros.

The format of the DEFINE statement is:

```
DEFINE name value
```

For more information, see the section "DEFINE Statement," earlier in this chapter.

Data Types

ELF supports five types of data: numbers, strings, arrays, and binary objects, and NULL.

Numbers

Numbers are stored in ELF as 16-digit, double-precision, floating-point values.

If a number is larger than six digits, ELF displays it in scientific notation. Arithmetic calculations are performed with full precision. When a number is converted to a string, the converted string has a maximum of six digits of precision.

When specifying numbers, you can use regular notation or scientific notation. For example, you can specify a number as 290000000 or as 2.9e+8.

You can use IS_NUMBER@ to determine if the value of a variable is a number.

Strings

An ELF string value is a sequence of ASCII characters. The only constraint on the size of a string is the amount of memory that exists on your system. String values must always be enclosed in double quotes as in the following example:

```
name = "string"
```

You can include double quotes within a string by preceding them with a back slash (\). For example:

```
"the answer \"no\" is correct"
```

You can include a back slash in a string by preceding it with a back slash (\ \).

You can use IS_STRING@ to determine if the value of a variable is a string.

Refer to the section "String Variables" for more information.

New Line Notation

You can indicate that a new line should be inserted in an INFO_MESSAGE@ string using the notation "\n" to specify the new line. When a new line symbol is encountered in a string used by INFO_MESSAGE@, a new line is inserted and the remainder of the string is placed on the next line.

For example, the macro

```
INFO_MESSAGE@("Name is invalid\nPlease type another
              name")
```

displays the following message in a dialog box:

```
Name is invalid
Please type another name
```

Tabs and Spaces

To add special characters to a string, you can enter the character in your string by using the following notation:

\(*num*)

where num is the ASCII representation of the string. A tab character, in this notation, is an ASCII 9, and a space character is an ASCII 32. For example:

```
macro test
    info_message@("hello\(9)\(9)hello")      ' Two tabs are added
end macro
```

Arrays

An ELF array is a group of elements consisting of numbers, strings or other arrays. An array has no fixed size. The array size increases as you assign values to the array. An ELF array can contain as many elements as memory can hold.

Arrays in ELF are *heterogenous*. A heterogenous array can contain more than one type of data. For example, the following code sample defines an array that contains both numbers and strings:

```
Macro array_test
    var array_variable
    array_variable = 1, "two", "three", 4
    dump_array@(array_variable)
endmacro
```

ELF arrays are zero-based. This means that the value that occupies the first position in an ELF array is in position 0. In the array_test macro, array_variable[0] = 1, array_variable[1] = "two", and so on.

You can use IS_ARRAY@ to determine if a variable is an array. Use ARRAY_SIZE@ to determine the current size of an array.

Binary Objects

A binary object contains bytes of data. Binary objects are useful for representing information that is manipulated or analyzed byte by byte. For example, you can convert a file to a binary object for transfer to a machine that might not be able to interpret other file formats.

You can use IS_BINARY@ to determine if the value of a variable is a binary object. There are also a collection of macros for interpreting and manipulating binary data.

Null Datums

The null datum is the value of a variable that has not been assigned a value. With a macro call that accepts optional arguments, any argument that is not supplied has the value of the null datum. The pre-defined constant NULL is used to represent the null datum.

You can use IS_NULL@ to determine if the value of a variable is NULL.

DEFINE Statement

The DEFINE statement is used to assign names to constants. This often enhances the readability of your code. The format of the DEFINE statement is:

```
DEFINE name value
```

For example, you could assign the name OFF to the constant value 0 as follows:

```
DEFINE OFF 0
```

Now, whenever the name OFF is used in your macro, it is interpreted by ELF to be the value 0. For example:

```
VAR switch_set
switch_set = OFF
```

The following macro uses two DEFINE statements.

```
DEFINE Children 2
DEFINE Income 20000

macro College

var money

/*
*     The Random_Seed@ macro generates a random integer from 0 to 32767
*/
      money = Random_Seed@()+ Income
/*
*     The two IF statements below use a variable (money) and a constant (children)
*/

      if money/children > 20000 then info_message@("You are going to college!")

      if money/children < 20000 then info_message@("You need a scholar ship!")
endmacro
```

DEFINE statements are often used in ELF header files to assign names to error codes, such as:

```
DEFINE ERR#WRONG_WINDOW 206
```

DEFINE statements are usually placed at the start of the macro document, before any macros. The DEFINE statements only apply to the macro document in which they reside.

DEFINE statements are often put in a header file, and included in macro documents through the INCLUDE statement.

Refer to the sections "INCLUDE Statement", "ELF Include Files", and "Constants" for related information.

ENDMACRO Statement

The ENDMACRO statement is the last statement in an ELF macro. ENDMACRO takes no arguments.

Refer to the section "MACRO Statement" for related information.

EQV Operator

EQV is both a logical operator and a bitwise operator.

As a logical operator, EQV is used primarily in conditional expressions. For an expression containing an EQV operator to evaluate to TRUE, both of the expressions must be either TRUE or FALSE. A truth table for EQV operator follows:

EQV	**0**	**1**
0	1	0
1	0	1

Note that:

EQV = Not XOR

EQV can also be used for bitwise operations. For example, the following statement displays the number -4 in an info box.

```
info_message@(5 EQV 6)  '00000101 AND 00000110 = 11111011 =-4
```

Refer to the section "Operator Precedence" for related information.

EXTERN Statement

Use the EXTERN statement to reference a global variable that is used in the current macro document, but is declared in a different installed macro document.

For example, if you declare the global variable Inflation in the macro document named Budget and then use the Inflation variable in the macro document named Projections, you must reference the Inflation variable in the Projections document using an EXTERN statement.

An EXTERN statement can be placed at any point in a macro document, but it must appear prior to any statement that references the global variable. Once a global variable has been referenced using an EXTERN statement, any macro in the macro document can reference that variable. The format for an EXTERN statement is:

EXTERN variable_name[, variable_name, variable_name...]

For example, in the following statement the global variables Orders, Inventory, and Units are referenced:

```
EXTERN Orders, Inventory, Units
```

FOR Loop

The FOR and NEXT statements create a FOR loop that executes a series of statements a specified number of times.

The format of a FOR loop is:

```
FOR variable=expression TO stop_expression [STEP
        expression]
loop statements
NEXT variable
```

A FOR loop is a convenient method for loading data into an array. In the following macro, the numbers 1 to 100 are loaded into the Numbers array.

```
macro loadarray
var loop_counter, Numbers

for loop_counter = 1 to 100                ' loop counter is set to 1 for the first pass
                                           ' through the loop, then incremented by
                                           ' 1 for each successive pass. In this example, the
                                           ' loop will be executed 100 times.

/*
*     loop_counter provides the index to the Numbers array and the value loaded into
*     the array.
*/

Numbers[loop_counter] = loop_counter

next loop_counter                          ' The NEXT provides an ending statement for
                                           ' the FOR loop.

dump_array@(Numbers)          ' Displays the contents of the numbers array
endmacro
```

When ELF encounters a FOR statement, it assigns the value of an expression to a variable. It then compares this variable value to a stop

expression. If the variable value is greater than the stop value, ELF resumes execution of the macro at the line following the NEXT statement. If the variable value is less than the stop value, ELF executes the loop statements.

When ELF encounters the NEXT statement, it increments the variable by the STEP value. The default STEP value is 1. ELF then returns to the FOR statement and re-evaluates it. The next section describes the STEP statement in more detail.

NEXT STEP Statement

The NEXT STEP statements tells ELF to skip the remainder of the statements within the loop's body and resume execution at the top of the FOR loop. The syntax for this statement is:

```
NEXT STEP variable
```

where variable is the counter variable of the loop. The following example shows the NEXT STEP statement.

```
FOR i = 0 to 99
    ...
    if COND = 1
            NEXT STEP i

/*
*   If COND = 1, these statements are skipped.
*/
    info_message@("Condition is not 1")    ' This statement executes if COND <> 1

NEXT i          'Execution resumes here of COND = 1. i is incremented, and the
                ' Loop continues.
```

STEP Statement

You can include an optional STEP value if you do not want to use the default value of 1. If you provide a negative STEP value, the loop statements are performed until the variable value is less than the stop value. The following code has two FOR loops. The first loop loads the even numbers from 1 to 100 into the Numbers array. The second loads the even negative numbers from 0 to -100 into the Numbers array.

```
macro loadarray
var loop_counter, Numbers, x
/*
*  Loop counter is set to 2 for the first pass through the loop, then incremented
*  by 2 for each successive pass. In this example, the loop will be executed
*  50 times.
*/
for loop_counter = 2 to 100 STEP 2    '
   x = x + 1                                  ' x is used as the array index.
   Numbers[x] = loop_counter

next loop_counter                                    ' The NEXT provides an ending statement
                                              ' for the FOR loop.

for loop_counter = -2 to -100 STEP -2
/*
*  Loop counter is set to -2 for the first pass through the loop, then decremented
*  by 2 for each successive pass. In this example, the loop will be executed 50
*  times.
*/
x = x + 1                                     ' x is used as the array index.
Numbers[x] = loop_counter
next loop_counter                             ' The NEXT provides an ending statement for the
                                    ' FOR loop.
dump_array@(numbers)
endmacro
```

Refer to the sections "BREAK Statement" and "NEXT STEP Statement" for related information.

FORMAT Statement

Use the FORMAT statement to define *format variables.* A format variable is a type of array variable that describes the structure of the data. Using a format variable, you can assign names to all or part of an array.

Formats are defined using the FORMAT statement as follows:

```
FORMAT format_name
    element0_name,
    element1_name,
    element2_name,
    ...
    elementN_name
```

format_name is a name that identifies the format. element_name items are the names by which you want to refer to the format's elements. Each element name in the FORMAT statement is separated by a comma.

For example, suppose you want to define a format that contains a "profit" value for each of four fiscal quarters and for the year:

```
FORMAT budget_info
    first_qtr_profit,
    second_qtr_profit,
    third_qtr_profit,
    fourth_qtr_profit,
    year_profit
```

After defining the format, you can define a variable that uses it as follows:

```
VAR FORMAT template_name variable
```

For example, you can apply the budget_info format to the variable budget96 as follows:

```
VAR FORMAT budget_info budget96
```

When a format is applied to a variable, you can use the element names defined in it to refer to array elements. For example, you could use the format names to assign values to the variable as follows:

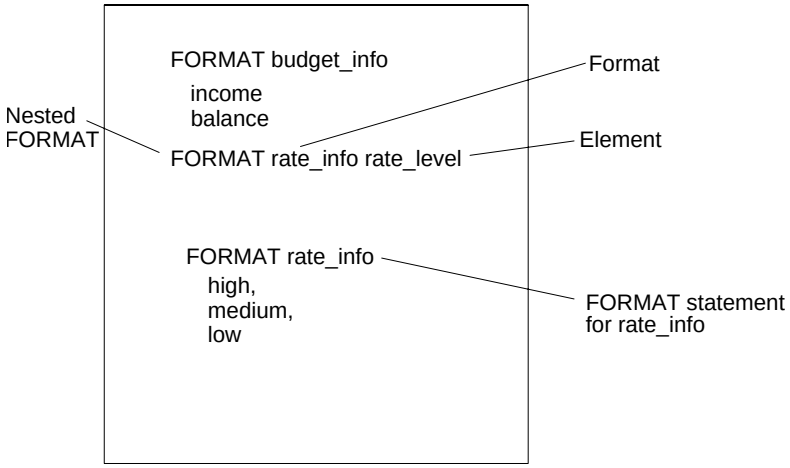
```
variable.element_name=expression
```

For example:

```
budget96.first_qtr_profit = 22534.89
budget96.second_qtr_profit = 12345.00
budget96.third_qtr_profit = -10234.90
budget96.fourth_qtr_profit = 35897.50
budget96.year_profit = 60542.49
```

Nesting FORMAT Statements

An element within a formatted variable can also be a formatted variable. That is, you can nest FORMAT statements to define sub-formats, as shown in the following:



You assign values to sub-array elements using the following format:

```
variable.element_name.sub_element_name = expression
```

For example:

```
VAR FORMAT budget_info budget96
budget96.rate_level.high = 12.5
budget96.rate_level.medium = 11.75
budget96.rate_level.low = 10.865
```

All format elements and sub-format elements can be referenced by stringing together the array variable names, each separated by a period (.), and adding the element name to the end.

Function Statement

A FUNCTION is a collection of ELF statements whose purpose is to return a numerical value. For example, you might define a FUNCTION that returns a sum of numbers.

A FUNCTION is formatted as follows:

```
FUNCTION name(arguments)
    statements
    RETURN(value)
ENDFUNCTION
```

Function Names and Arguments

The name of a FUNCTION must be unique within the ELF task, and must adhere to the following naming conventions:

- Names must begin with a letter of the alphabet.
- Names can contain alphabetic, numeric, or underscore (_) characters.
- Names cannot include spaces.

Function names must not include:

- the "#" character.
- The @ symbol, $ symbol, or _ (underscore) symbol as the last character in the macro name. By convention, only ELF built-in macros can use these symbols as the last character in the macro name.

The arguments are an optional list of arguments that are passed to the function.

Recording Functions

A call to a function is not recorded by the Keystroke Recorder. This is the only important difference between a FUNCTION and a standard macro.

By convention, a function returns a number to the calling macro, but ELF does not enforce this convention.

Refer to the section "RETURN Statement" for additional information on returning information to a calling macro.

The following code shows a simple macro that calls a function.

```
macro Call_function
var x, total, add_to_total, function_argument

for x = 1 to 10
      function_argument = x
      add_to_total = Double_It(function_argument)        ' Calls the Double_It function
      total = total + add_to_total
next x

info_message@(total)                                     ' The total is 110.
endmacro

/*
*     The Double_it function accepts a  number from the calling macro, doubles it,
*     and returns the result.
*/
Function Double_It(x)

x = 2*x
return(x)

endfunction
```

Global Variables

Global variables are defined independently of ELF macros and may be referenced by any macro within a given task. A global variable retains its value for the duration of a task.

Global variables are declared using VAR declarations that are placed at the start of the macro document, outside of any macro or function. For example, the following declares Inflation as a global variable:

```
VAR Inflation
MACRO Budget90
    Inflation = 5
    ....
    statements
    ....
ENDMACRO
```

When a task is initiated, the value of all global variables in the task is the null datum.

Use EXTERN to reference a global variable that is used in the current macro document, but is declared in a different installed macro document.

Global variables are only global to a single ELF task. If you want to exchange variable information between ELF tasks use a system variable.

Refer to the sections "System Variables" and "EXTERN Statement" for related information.

GOTO Statement

A GOTO statement is used to branch to a specific location in your document. The format of a GOTO statement is:

```
GOTO label
```

The label is a text string that appears directly before the statement you wish to branch to. The label is followed by a colon. The following example shows the GOTO statement.

```
macro Beggar
var answer
ask_again:
    answer = prompt@("Can I borrow a quarter?")
    if answer <> "yes" Goto ask_again
endmacro
```

IF Statement

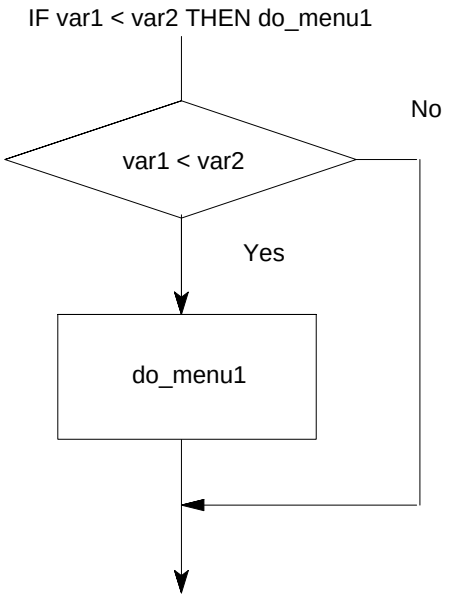
IF-THEN and IF-THEN-ELSE statements specify what statements in an ELF macro are executed based on the evaluation of an expression.

An IF-THEN statement is formatted as follows:

```
IF expression THEN statement
```

If the value of expression is not FALSE (a value other than 0), ELF executes the statement. If the value of expression is FALSE, then the statement is not executed and processing continues with the next statement after the IF-THEN statement.

The expression can compare values, such as IF var1 > var2, or it can be a value, such as IF var. The keyword THEN is optional. The following shows an IF-THEN statement:



In the following example, ELF throws an error if the expression (column < 0) evaluates to TRUE. If the expression evaluates to FALSE, ELF continues processing at the statement IF (column > 26).

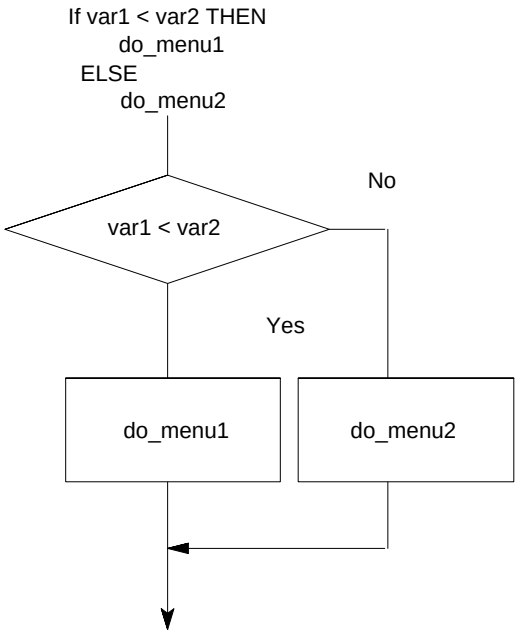
```
IF (column < 0)
    ERROR@(99, "Not a valid column")
IF (column > 26) ...
```

IF - THEN - ELSE

An IF-THEN-ELSE statement is formatted as follows:

```
IF expression THEN statement1 ELSE statement2
```

With an IF-THEN-ELSE statement, statement1 is executed if expression evaluates to TRUE. Otherwise, statement2 is executed. The following shows an IF-THEN-ELSE statement.



The ELSE operation in an IF-THEN-ELSE statement is typically another IF-THEN-ELSE statement, as shown in the following example:

```
IF result = "S"
    menu_demo1
ELSE IF result = "F"
    menu_demo2
ELSE IF result = "V"
    menu_demo4
```

See the sections "CASE Statement" and "Conditional Statements" for related information.

IMP Operator

IMP (implication) is both a logical operator and a bitwise operator.

As a logical operator, IMP is used primarily in conditional expressions. For an expression containing an IMP operator to evaluate to FALSE, The first expression must be TRUE, and the second expression must be FALSE. Otherwise the expression containing an IMP operator is TRUE.

A truth table for IMP operator follows:

IMP	**0**	**1**
0	1	1
1	0	1

IMP can also be used for bitwise operations. For example, the following statement displays the number -2 in an info box.

```
info_message@(5 IMP 6)  ' 00000101 AND 00000110 = 11111110 = -2
```

Refer to the section "Operator Precedence" for related information.

INCLUDE Statement

For a macro document to use information in a header file, the header file must be included in the macro document. The INCLUDE

statement is used to include a header file in a macro document. The format for the INCLUDE statement is:

```
INCLUDE "filename"
```

where filename is the name of the header file you want to include. filename must be enclosed in quotation marks and should include the file name extension. For example, to include the ELF include file Words_.am, use the following line:

```
INCLUDE "words_.am"
```

When an INCLUDE statement is encountered, ELF looks for the header file in three directories, in this order:

1. Your axhome/macros directory.
2. ax/axexec/axlocal. This is your Applixware install directory.
3. ax/axexec/axdata/elf directory. This directory contains the ELF header files.

If the header file is not in any of these directories, then you must use a full path name to indicate the directory in which the header file is placed.

All INCLUDE statements should be placed at the beginning of the macro document. It is also recommended that header files contain only DEFINE, FORMAT, and EXTERN statements. If you include a pound sign (#) before your INCLUDE statement, it is ignored. For example:

```
#INCLUDE "wp_.am"     ' This line works - the # is ignored.
```

NOTE: ELF does not support nested include files. INCLUDE statements in ELF include files cause compiler errors.

Local Variables

A local variable is defined within an ELF macro and can only be referenced by that macro. To declare a local variable, you include the VAR statement between the opening statement of the macro (MACRO, UIMACRO, FUNCTION) and the terminating statement of the macro (ENDMACRO, ENDFUNCTION). The following example declares the local variables contents and file.

```
FUNCTION Ascii
    VAR contents, file

    file =prompt@ ("Name of file")
    contents=Read_Ascii_File@(file)
    RETURN(contents)
ENDMACRO
```

When a macro is called, all locally declared variables have the null datum as an initial value.

Refer to the sections "Variables", "Global Variables" and "System Variables" for related information.

Logical Operators

An operator is the part of an expression that specifies the type of operation or calculation to be performed. Logical operators are used to compare two expressions in ELF:

The following expression contains a logical operator:

```
if X =10 and Y = 5  statement
```

The expression instructs ELF to compare the variable X with 10, and compare the variable Y with 5. and act on the ensuing statement if both X = 10 and Y = 5 are TRUE. The word and is a logical operator.

ELF supports arithmetic, relational, and logical operators. The following table lists all the ELF logical operators. The column "Leading/Trailing Spaces" indicates whether you are required to include spaces before and after the operator.

Table 4-3 Logical Operators

Logical Operator	Operation	Example	Leading/Trailing Spaces
Not	Inverse	NOT a=c	Required
AND	And	a>b AND a>c	Required
OR	Or	a>b OR a>c	Required
XOR	Exclusive or	a>b XOR a>c	Required
EQV	Equivalence	a-b EQV a-c	Required
IMP	Implication	a-b IMP a-c	Required

Refer to the section "Operator Precedence" for related information.

Macro Statement

The MACRO statement is the first statement in an ELF macro. The MACRO statement is always followed by a string, which is the name of the ELF macro.

Macro Names and Arguments

The name of a macro must be unique within the ELF task, and must adhere to the following naming conventions:

- Names must begin with a letter of the alphabet.
- Names can contain alphabetic, numeric, or underscore (_) characters.
- Names cannot include spaces.

Variable and macro names must not include:

- the "#" character.
- The @ symbol, $ symbol, or _ (underscore) symbol as the last character in the macro name. By convention, only ELF built-in macros can use these symbols as the last character in the macro name.

The arguments are an optional list of arguments that are passed to the macro. The argument list is a set of values passed from another ELF macro. If more than one argument is passed from another macro, the arguments are separated by commas:

```
macro Example(arg1, arg2)
```

The following example shows two ELF macros. Test passes a string to the macro StringLength. Note that the MACRO statement for StringLength contains the argument passed from test.

```
macro Test

var astring, length

    astring = "Hello, this is a string."
    length = StringLength(astring)          ' Call the StringLength Macro with astring
                                                ' as an argument.
    info_message@("The length is "++length)
endmacro
```

```
macro StringLength(str)
/*
*       The str variable does not have to be declared here because
*       it was declared in the test macro.
*/

        return(len@(str))       ' Return the length of the string

endmacro
```

Refer to the section "ENDMACRO Statement" for related information.

NOT Operator

NOT is both a logical operator and a bitwise operator.

As a logical operator, NOT is used primarily in conditional expressions. For an expression containing a NOT operator to evaluate to TRUE, the object in the expression must be FALSE.

NOT can also be used for bitwise operations. For example, the following statement displays the number -7 in a message box.

```
info_message@(NOT 6)  ' NOT 00000110 = 11111001 = -7
```

The following example is an IF statement using a NOT operator. The GOTO executes if the expression x=y is FALSE.

```
IF NOT x = y
{
Goto NextCompare
}
```

Refer to the section "AND Operator" for related information.

NOTHING Statement

The NOTHING statement performs no operation. It can be used in an IF statement as shown:

```
IF answer = "no"
    NOTHING
ELSE
    info_message@("Thank You")
```

In this case, ELF executes a statement only if an expression evaluates to FALSE.

The semi-colon (;) symbol can be used in place of the NOTHING statement. For example:

```
IF answer = "no"
    ;
ELSE
    info_message@("Thank you!")
```

ON ERROR Statement

The ON ERROR statement loads a new error handler onto the execution stack of the current macro. Error handlers are run when an error is thrown by the ELF macro.

The following sample code shows the structure of an ELF error handler. This error handler checks whether the error code is 45, and if it is, the error is displayed in an error message box and processing is

sent to the again label. If the error is not error code 45, it is rethrown so that another error handler or the ELF error handler can handle the error.

```
ON ERROR
{
   IF ERROR_NUMBER@() = 45
   {
            ERROR_BOX@
            GOTO again
   }
   ELSE
            RETHROW_ERROR@
}
```

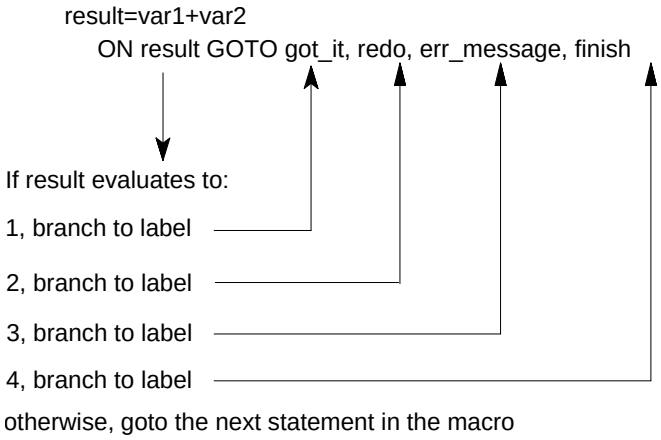
For more information about error handling, see Chapter 5, "Error Handling," in this manual.

ON GOTO Statement

The ON GOTO statement causes your macro to branch to a label indicated by the computed value of an expression. The format of a computed GOTO statement is:

ON expression GOTO label, label, label...

When ELF encounters a computed GOTO statement, it evaluates the expression and then branches to the label that occupies the position in the GOTO statement list corresponding to the computed value. The GOTO statement list is 1-based. The following figure illustrates the way in which computed GOTO statements work.



For example, if the computed value of an expression is 3, ELF branches to the third label listed in the GOTO statement. In the following statement, ELF would branch to the label err_message since the value of the expression is 3:

```
result=3
ON result GOTO got_it, re_do, err_message, finish
.....
got_it:
        statements
        ...
re_do:
        statements
        ...
err_message:
        statements
        ...
finish:
        statements
        ...
```

If the computed value of an expression is 0, or the value exceeds the number of labels in the GOTO statement list, ELF does not branch out of the regular statement sequence; processing continues with the next statement following the ON GOTO statement. If the value is less than 0 or greater than 255, ELF throws an error.

If possible, you should avoid using computed GOTO statements. It is often difficult to follow the flow of a program when computed GOTO statements are used, so maintaining macros that contain such statements can be cumbersome.

Operator Precedence

The order in which operations in an ELF expression are performed is determined by rules of precedence. These rules are:

- Macro calls are performed first.
- Arithmetic operations are performed second, in the following order:

  1. arithmetic negation
  2. exponentiation
  3. multiplication or division (in order from left to right)
  4. integer division
  5. remainder operation (MOD)
  6. string concatenation
  7. addition or subtraction (in order from left to right).

- Relational operations are performed third. All relational operations share the same order of precedence.
- Logical operations are performed last, in the following order:

  1. Inverse (NOT)

2. All other logical operations share the same order of precedence.

Operations that share the same order of precedence are performed in the order in which they are encountered (left to right).

You can change the order in which operations are performed using parentheses. Expressions contained in parentheses are evaluated first starting with the innermost parentheses. For example:

```
2+3*4+5=19
2+(3*4)+5=19
(2+3)*(4+5)=45
```

OR Operator

OR is both a logical operator and a bitwise operator. As a logical operator, OR is used primarily in conditional expressions. For an expression containing an OR operator to evaluate to TRUE, either of the objects in the expression must be TRUE.

OR can also be used for bitwise operations. For example, the following statement displays the number 7 in an information box.

```
info_message@(5 AND 6)  ' 0101 OR 0110 = 0111
```

The following example is an IF statement that tests an expression, and executes two statements if the tested expression is TRUE.

```
IF Buy_Applixware OR Buy_Applix_Stock

{

Bank_Account = Bank_Account + 1000000
Goto EARLY_RETIREMENT

}
```

Refer to the "AND Operator" section for related information.

Reserved Words

The following reserved words cannot be used as macros, functions, labels, or variable names:

AND	ARRAY	ARRAYOF	ARRAYOF2
BEGIN	BREAK	CALL	CASE
CONTROL	DEFAULT	DEFINE	#DEFINE
ELSE	END	ENDCASE	ENDFUNCTION
ENDGET	ENDMACRO	ENDSELECT	ENDSET
EQV	ERROR	EXTERN	FALSE
FOR	FORMAT	FUNCTION	GET
GOTO	HELP	IF	IMP
INCLUDE	#INCLUDE	INTEGER	LET
MACRO	MOD	NEXT	NODEBUG
NOT	NOTHING	NRMACRO	NULL
NUMBER	OBJECT	OBJVAR	OF
ON	OR	POINTER	RETURN
SELECT	SELECT1	SET	STEP
STRING	STRUCT	SYSVAR	THEN
TO	TRUE	UIMACRO	VALUES
VAR	WEND	WHILE	XOR
@@@			

Relational Operators

An operator is the part of an expression that specifies the type of operation or calculation to be performed. Relational operators instruct ELF to compare two values. The following expression contains a relational operator:

```
if X < 5 statement
```

The expression if X < 5 instructs ELF to compare the variable X with 5, and act on the ensuing statement if X is less than 5. The less than sign (<) is a relational operator.

ELF supports arithmetic, relational, and logical operators. The following table lists all the ELF relational operators. The column "Leading/Trailing Spaces" indicates whether you are required to include spaces before and after the operator when you type the operator.

Table 4-4 Relational Operators

Relational operator	Operation	Example	Leading/Trailing Spaces
<	Less than	1<2	Optional
>	Greater than	2>1	Optional
<=	Less than or equal	1<=2	Optional
>=	Greater than or equal	2>=1	Optional
=	Equal to	2=2	Optional
<>	Not equal to	1<>2	Optional
<<	Less than (string)	"Arthur"<<"Barnum"	Optional

Table 4-4 Relational Operators (cont.)

Relational operator	Operation	Example	Leading/Trailing Spaces
>>	Greater than (string)	"Handley">>"Evans"	Optional
<<=	Less than or equal (string)	"Clark"<<="Evans"	Optional
>>=	Greater than or equal (string)	"Evans">>="Clark"	Optional

Return Statement

When RETURN is encountered, the macro or function is terminated and no other statements in the macro are executed. RETURN statements are only necessary when you want to exit a macro before the usual processing of the macro is completed. For example, your macro might include an IF-THEN statement in which the macro is terminated if a certain condition is met. If no RETURN is found in a macro or function, the macro or function is exited when the ENDMACRO statement is encountered.

There are two forms of RETURN statements. One returns to the calling macro. The other returns a specified value.

The format of a RETURN statement that does not return a specified value is:

```
RETURN
```

The format of a RETURN statement that returns a specified value is:

```
RETURN(expression)
```

The expression is any ELF expression. Use this RETURN statement in a function that returns a value to its calling macro. For example, the following function returns a string if a certain condition is met.

```
FUNCTION BudgetAnalysis(CurrentVal)
    VAR actual_val, budgeted
    budgeted = PROMPT@("Enter amount budgeted for this
                item")
    actual_val = budgeted - CurrentVal
    RETURN(actual_val)
ENDMACRO
```

Since, by definition, all ELF macros return a value, the RETURN statement is the same as RETURN(NULL).

String Variables

String variables are a flexible and powerful part of ELF. To assign a string to a variable, set the variable name equal to the string using parentheses to delimit the string:

```
name = "Queen of Sheba"
```

String Concatenation

To concatenate two strings, use the ++ operator. The following macro displays "Hello Applix" in an information box.

```
Macro Hello
    info_message@("Hello "++"Applix")
endmacro
```

String Compares

You can compare strings using the >> (greater than) and << (less than) operators. Characters are compared based on the ASCII value of the character. The following lists basic characters in ascending order from left to right:

0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

The following macro tests two strings to see which is greater.

```
macro teststrings
var string1, string2
      string1  "First String"
      string2 = "Second String"
      if string1 >> string2
                  info_message@("The first string is greater")
      if string2 >> string1
                  info_message@("The second string is greater")
endmacro
```

Character Notation

To specify a special character within a string, you can express the character by its ASCII numeric value using the following notation:

```
\(num)
```

For example, the tab character is ASCII 9 and the space character is ASCII 32:

```
data = "The\(32)quick\(9)brown fox"
```

Prints as follows:

```
The quick    brown fox
```

Executable String Variables

If a variable is assigned a string value, and that string is the name of an executable macro or function, the variable is executable. You indicate that a string variable should be executed by preceding the variable name with an exclamation point (!).

For example, suppose you declare a variable and assign it a string value as follows:

```
VAR check
check = "DIR_EXISTS@"
```

You can execute the check variable by preceding it with an exclamation point:

```
!check("/usr/sam")
```

In this example, !check is interpreted as DIR_EXISTS@, so the function DIR_EXISTS@("/usr/sam") is executed.

Executable string variables are useful in instances when the choice of what macro to run is dependent on whether certain conditions are met. In the following example, the macro that is run depends on the response typed at a prompt.

```
MACRO start_app
   VAR answer, do_macro
try_again:
   answer = PROMPT@("Number of program to run")
   IF answer = "1"
      do_macro = "budget"
   ELSE IF answer = "2"
      do_macro = "transactions"
   ELSE IF answer = "3"
      do_macro = "year_to_date"
   ELSE
   {
      INFO_MESSAGE@(
         "Invalid response.  Please type another number")
      GOTO try_again
   }
```

```
    !do_macro
ENDMACRO
```

In this example, the statement !do_macro indicates that the macro budget, transactions, or year_to_date is run depending on the value of the answer variable.

Refer to the section "Variables" for more information.

System Variables

System variables are variables that retain their value throughout the Applixware environment. For instance, you could assign a value to a system variable used within a dialog box macro and then use the value from the dialog box macro in a different application. ELF supports two different system variable methods. The first declares variables in a way similar to the way you declare global variables. For example, here is the declaration of system variable named BUDGETinfo:

```
SYSVAR BUDGETinfo
```

The second method, which uses ELF macros, is discussed in the next section.

If your application is contained within more than one source file, place this declaration in each of the source files. (The best way to do this is to place the declaration in a header file, then include the header file in all of your source files.)

Unlike global variables, you do not have to use the EXTERN keyword to tell ELF that the variable is defined in another file.

After you define a system variable, it remains defined until the end of your Applixware session. That is, the memory allocated to the variable

remains in use even after the task that uses the memory stops executing.

Because the memory used by a system variable remains allocated during the current session, you should be careful not to overuse this feature by placing large amounts of data into system variables. If you need to use a system variable that contains a lot of data, you can release its memory space back to ELF by assigning the variable a NULL value when you are done with the information.

Case Sensitivity with System Variables

System variable names are case sensitive when you are referencing them with the SYSTEM_VAR@ and SET_SYSTEM_VAR@ ELF macros. System variable names declared using the SYSVAR declaration are always stored in ELF in uppercase. An example follows:

```
 SYSVAR mySysVar                ' This variable is stored as MYSYSVAR -
                                ' all uppercase.
macro Test_System_Variables
      mysysvar = 10                                  ' MYSYSVAR = 10
      set_system_var@("MYSYSVAr", 12)                ' MYSYSVAr =  12
      INFO_MESSAGE@(SYSTEM_VAR@("MYSYSVAr") )        ' This returns 12
      info_message@(SYSTEM_VAR@("MYSYSVAR"))         ' This returns 10
endmacro
```

You can declare system variables with the SYSVAR keyword, or through the SYSTEM_VAR@ ELF macro. The SYSTEM_VAR@ ELF macro allows you to declare system variables in mixed cases, while all variables declared with SYSVAR are uppercase.

The following lists the rules for declaring system variable with and without the SYSVAR declaration:

- sysvar foo is same as SYSVAR FOO
- sysvar foo is same as system_var@("FOO")

- sysvar foo is NOT same as system_var@("foo")

SET_SYSTEM_VAR@

SET_SYSTEM_VAR@ sets the value of a system variable. The format for the SET_SYSTEM_VAR@ macro is:

```
SET_SYSTEM_VAR@(name,value)
```

name, a string, is the name of the system variable for which you want the value. If you declare a system variable as a SYSVAR declaration, name must be all uppercase to access that system variable.

SYSTEM_VAR@

SYSTEM_VAR@ returns the value of a system variable. The format for the SYSTEM_VAR@ macro is:

```
value = SYSTEM_VAR@(name)
```

name, a string, is the name of the system variable for which you want the value. If you declare a system variable as a SYSVAR declaration, name must be all uppercase to access that system variable.

UIMACRO Statement

If you create a macro that calls a dialog box, you may want to designate the macro as a user interface macro. The difference between a standard ELF macro and a user interface macro is the way in which the macro is represented in a keystroke recording.

User interface macros start with the UIMACRO keyword. For example:

```
UIMACRO name(arguments)
    statements
ENDMACRO
```

The UIMACRO statement is the first statement in an ELF macro. The UIMACRO is always followed by a string, which is the name of the ELF macro.

Macro Names and Arguments

The name of a macro must be unique within the ELF task, and must adhere to the following naming conventions:

- Names must begin with a letter of the alphabet.
- Names can contain alphabetic, numeric, or underscore (_) characters.
- Names cannot include spaces.

Macro names must not include:

- the "#" character.
- The @ symbol, $ symbol, or _ (underscore) symbol as the last character in the macro name. By convention, only ELF built-in macros can use these symbols as the last character in the macro name.

The arguments are an optional list of arguments that are passed to the macro from the macro that calls it. If more than one argument is passed from another macro, the arguments are separated by commas:

```
UIMACRO Example(arg1, arg2)
```

Recording User Interface Macros

When you make a keystroke recording of a macro and the recording contains calls to a dialog box, the dialog box call is recorded in one of two ways:

- When the macro that calls the dialog box is a user interface macro, only the call to the macro that displays the dialog box is recorded. (In most cases, this is the macro DB_DISPLAY@.) When you replay the recording, the dialog box is displayed and you must enter information into the dialog box and exit the dialog box in order to continue replaying the recording.

- When the macro that calls the dialog box is a standard macro, calls to the macros that perform the actions specified by dialog box selections are recorded. When you replay the recording, the dialog box is *not* displayed. The same actions performed as a result of dialog box selections made at the time of the recording are performed when you replay the recording.

Except for the UIMACRO keyword and the way in which the macros are recorded, there is no difference between the way regular macros and user interface macros work or the way you create them.

All the menu options in Applixware applications that call dialog boxes are standard macros. When a recording is made involving an Applixware menu option that calls a dialog box, the underlying dialog box calls are recorded. When you replay the recordings, no dialog boxes are displayed.

There is no need to provide a distinction between regular and user interface macros when you create macros that do not involve dialog box calls. Therefore, you should define macros that do not involve dialog box calls as standard macros.

Variables

Variables are named data units that contain values. A variable may take the form of a string, number, array, macro, or function.

ELF variables must be declared before they are assigned values or referenced. ELF variables have no preset type. An ELF variable can be, at any point in time, a string, number, array, macro, or null datum. When an ELF variable is declared, its initial value is the null datum.

Scope

Scope refers to the accessibility of a variable. The scope of a variable is determined by the location of the variable declaration in an ELF macro. ELF supports local, global, and system variables. These variable types represent three different levels of scope.

- A local variable has the most limited scope. A local variable is accessible only from the macro in which it is declared. Local variables are declared inside the body of the macro itself.
- A global variable is one that can be used by any macro for the duration of a particular ELF task. If a global variable is assigned a value in one macro contained within a particular task, that value can be used by other macros that are run as part of the task.

  Global variables are made available to other macros in the same task through the EXTERN statement.
- A system variable has the widest scope available in ELF. It can be used by any ELF macro for the duration of the Applixware session. System variables are declared through the SYSVAR declaration, or through the ELF macro SYSTEM_VAR@().

Declaring Variables

All variables must be declared before they are used in a macro. You declare variables using the VAR statement.

Assigning Values to Variables

Use assignment statements to assign a value to a variable. The value may be a number, string, array, or expression. The format of an assignment statement is:

```
variable = expression
```

For example:

```
charge = cost+(cost*.05)
```

An ELF variable assumes the data type of the value assigned to it. For example:

```
x = "12" 'x becomes a string variable
x = 12   'x becomes a numeric variable
```

For information on assigning values to array variables, see the section "Arrays" earlier in this chapter.

Refer to the sections "VAR Statement", "FORMAT Statement" and "VAR FORMAT Statement" for related information.

Passing Variables

Variables are always passed to macros and functions by reference. Because of this, constants that are passed to subroutines and functions cannot be changed In the example code that follows, the code on the

left passes a constant, and throws the error "not a variable." The code on the right shows the proper approach, passing a variable instead of a constant.

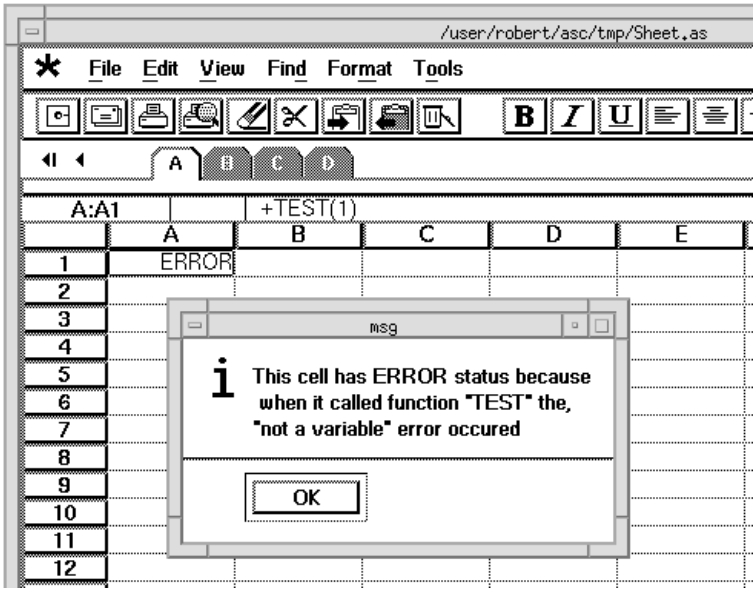
This Throws an Error

```
define greeting "Hello"

macro test

    test2(greeting)

endmacro


macro test2(hello)

    hello = hello++" joe"
    info_message@(hello)
    return

endmacro
```

This Works

```
define greeting "Hello"

macro test
        var x
        x = greeting
        test2(x)
endmacro

macro test2(hello)

        hello = hello++" joe"
        info_message@(hello)
        return

endmacro
```

The "not a variable" error can also occur when you try to pass a a numeric constant from a spreadsheet to a macro that tries to change that value. For example, suppose you ran the macro on the left of the preceding table like this:



VAR Statement

All variables must be declared before they are used in a macro. You declare variables using the VAR statement. The format for a VAR statement is:

```
VAR variable_name[, variable_name]...
```

You can declare more than one variable using one variable statement and you can use multiple VAR statements within a macro. If you include multiple variable names in a single VAR statement, the variable names must be separated by commas. The VAR declaration on the left is equivalent to the three VAR statements on the right:

VAR name, address, id	VAR name VAR address VAR id

The location of a VAR declaration determines whether a variable is a local variable or a global variable.

A variable's name cannot be longer than 60 characters.

Refer to the section "Variables" for related information.

VAR FORMAT Statement

You can define a variable that uses a previously-established FORMAT template as follows:

```
VAR FORMAT template_name variable
```

For example, you can apply the budget_info format to the variable budget96 as follows:

```
VAR FORMAT budget_info budget96
```

Refer to the section "FORMAT Statement" for related information.

Wend Statement

The WEND statement is used to terminate a WHILE loop. The structure of a WHILE loop is as follows:

```
WHILE expression
    loop statements
WEND
```

For more information on the WEND statement, see the section entitled While Loop, later in this chapter.

Refer to the following section "While Loops" for related information.

WHILE Loops

A WHILE loop executes a series of statements as long as the value of an expression is not false (a value other than 0). A WHILE loop is always terminated by a WEND statement. The format of a WHILE loop is as follows:

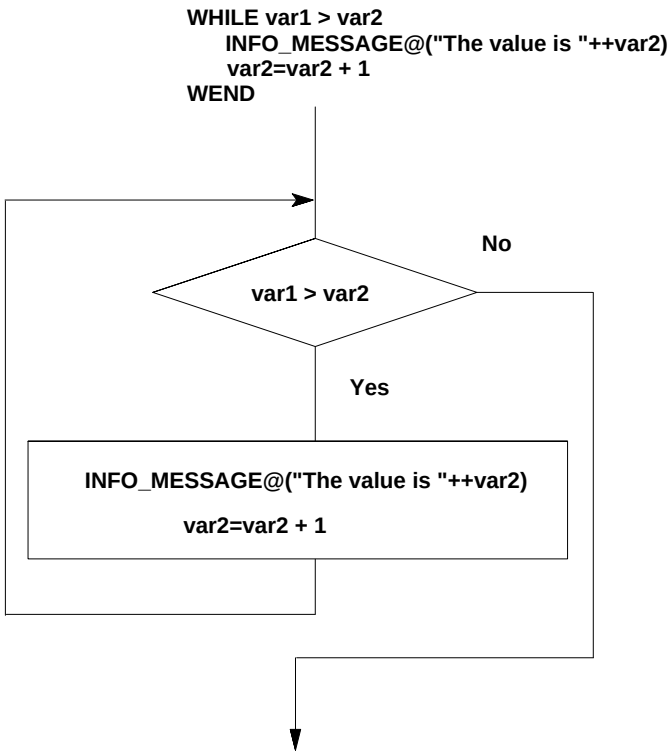
```
WHILE expression
    loop statements
WEND
```

ELF evaluates the expression following the WHILE statement. If it is not FALSE, ELF executes the statements in the WHILE loop in the order they appear.

The expression can be an expression that compares values, such as WHILE var1 > var2, or it can be a value, such as WHILE var. When

ELF encounters the WEND statement, it returns to the corresponding WHILE statement and re-evaluates the expression. If the expression remains not FALSE, the process is repeated. If the expression is evaluated to FALSE, ELF proceeds to the statement following the WEND statement.

The following illustrates the processing of a WHILE loop.



In the following example, ELF displays a message if the title of the current window is not equal to the variable my_name.

```
WHILE CURRENT_WINDOW_TITLE@() <> my_name
    INFO_MESSAGE@("Sort ready, select the spreadsheet
        again")
```

```
WEND
```

Loop Branching within a WHILE Loop

ELF allows you to prematurely terminate an iteration of a WHILE loop or the entire WHILE loop.

The NEXT WHILE statements tells ELF to skip the remainder of the statements within the loop's body and resume execution at the top of the WHILE loop. The syntax for this statement is:

```
NEXT WHILE
```

The BREAK statement tells ELF to skip the remainder of the statements with the loop's body and resume execution with the statement following the WHILE loop. The syntax for this statement is:

```
BREAK WHILE
```

Here is an example that uses both of these statements

```
WHILE (TRUE)
    ...
    if COND = -1
        BREAK WHILE
    if COND = 1
        NEXT WHILE
    ...
WEND
```

The WHILE statements control the loop branching within the WHILE loop. If the COND variable is set to -1, the loop is terminated. If COND is set to 1, the statements following the IF statement down to the WEND statement are skipped. For any other value of COND, the statements after the IF statements are executed.

The BREAK WHILE and NEXT WHILE can only cause a branch to occur within the current loop. That is, even if you nest a WHILE loop within a WHILE loop, you can only loop branch within the current WHILE loop. For example, you cannot break free of both loops with one BREAK statement.

XOR Operator

XOR is both a logical operator and a bitwise operator.

As a logical operator, XOR is used primarily in conditional expressions. For an expression containing an XOR operator to evaluate to TRUE, one of the objects in the expression must be TRUE, but not both.

XOR can also be used for bitwise operations. For example, the following statement displays the number 3 in an information box.

```
info_message@(5 XOR 6)  ' 0101 XOR 0110 = 0011=3
```

The following example contains an IF statement that tests x and y. If one of the variables is TRUE, but not both, the information message appears.

```
macro test
var x, y
y = 1
if x XOR y
    info_message@("Either X or Y is true, but not both!")
endmacro
```

5 The ELF Debugger

The ELF debugger allows you to interactively test your ELF program for logic errors. The debugger allows you to set break points, test variables during macro execution, and step through macro statements one at a time. This chapter covers the following topics:

- Debugger Mode
- Break Points
- Stepping through Macro execution
- Debugger Messages

Debugger Mode

You can use the ELF debugger to interactively follow the execution of a macro statement by statement. The debugger helps you discover any programming logic errors that exist.

To start debugger mode in the Macro editor, choose Tools → Debugger Mode.

While the Macro editor is in debugger mode, most of the Macro editor features are disabled. For example, you cannot type text while debugger mode is active. If you need to edit the macro, exit from debugger mode by choosing Tools → Debugger Mode again.

While in debugger mode, you can only select words in text. To select a word, click on it.

You can have up to 16 Macro Editor windows display the debugger mode window simultaneously. Displaying multiple debugger mode windows is useful, for example, when you have a macro that calls sub-macros and you want to follow the execution of both the parent macro and the sub-macros.

Debugger mode applies to the entire ELF environment, not just to the macro documents in which the debugger window is displayed. For example, if the debugger is displayed in one macro document window and you run a macro from a different Applixware window, debugging information for the macro is displayed in the macro document in which the debugger is displayed.

Running a Macro in the Debugger

There are two ways to run a macro in the debugger:

- Click on the macro name, then choose Run in the debugger window. The name of the macro does not have to be in the MACRO statement at the beginning of the macro. It can be anywhere, even in a different macro. For example, consider this set of macros:

```
macro test
var x
test2()
x = AXHOME_DIR@()
info_message@(x)
endmacro


macro test2
' This is called by the test macro
var x
info_message@("Hello")
endmacro
```

  To run the test macro in debugger mode, set the cursor in the word test in either of these two lines:

```
macro test
' This is called by the test macro
```

  When you press the run button, the test macro executes in debug mode. Note that the second line is a comment in test2.

- Choose ✶ → Run Macro and specify the macro to run.

Break Points

Before you run a macro you might want to set break points in the macro to cause the debugger to suspend execution of the macro.

While execution is suspended, you can perform various debugging tasks, such as displaying the current values of variables or moving up or down through the execution stack for the macro.

To set a break point in a macro, follow these steps:

1. Choose Tools → Debugger Mode to set Debugger mode on.
2. Place your cursor at the beginning of a line of executable code, and press Break. This message appears:

   break at line 5 in /user/applix/test.am

3. Run the macro.

When the macro is run (in debugger mode), execution is suspended immediately *before* the statement at which the break point is set.

Where to Set Break Points

Not all statements in an ELF macro are executable. Some statements, such as #include, are used by the ELF preprocessor. Other statements, such as variable declarations or comments, are used by the compiler. Table 5-1 shows a list of commonly-used ELF statements, and whether they support break points.

Table 5-1. ELF Statements and Break Points

Statement	Example	Break Point Allowed?
Macro	macro BigApp	N
Endmacro	ENDMACRO	Y
Variable Declaration	VAR myvariable	N
Conditional (IF)	If x = 1 { Goto Hello }	Y Y

Table 5-1. ELF Statements and Break Points (cont.)

Statement	Example	Break Point Allowed?
CASE of	CASE of x	Y
	statement	Y
CASE	CASE "teststring"	N
ENDCASE	ENDCASE	Y
Variable Assignment	X = 5 AString = "Applix"	Y
Label	JumpToHere:	N
#include	#include "errors_.am"	N
Function	Function MYFUNC	N
Endfunction	ENDFUNCTION	Y

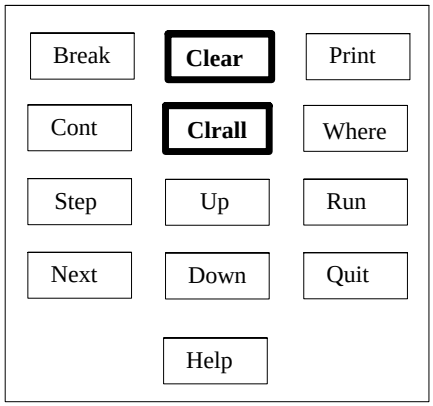
You can stop the execution of a macro before it is finished by pressing Quit while processing is suspended.

Note that while using the debugger you can run more than one macro at a time by running a new macro while the processing of one macro has been suspended. However, the debugger only follows the execution of a single macro at a time. If you click on Quit, the debugger terminates the current macro, not of all macros.

If your macro calls sub-macros, you can also set break points in the submacros if you want. If a break point is encountered in a sub-macro, the Macro Editor display is automatically updated to show the sub-macro. When execution of the sub-macro is completed, the Macro Editor display is automatically updated to show the calling macro again.

All breakpoints are cleared when you exit the debugger.

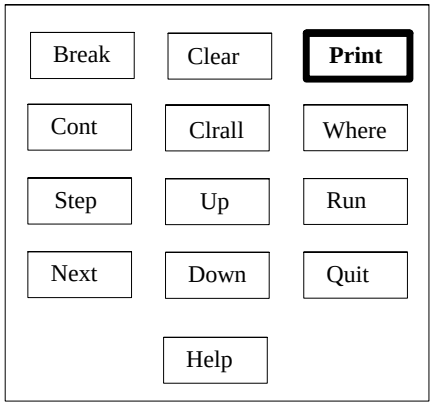
Clearing Break Points



To clear a single break point, click on Clear while at the break point.

To clear all break points currently set in all macro documents, click on ClrAll.

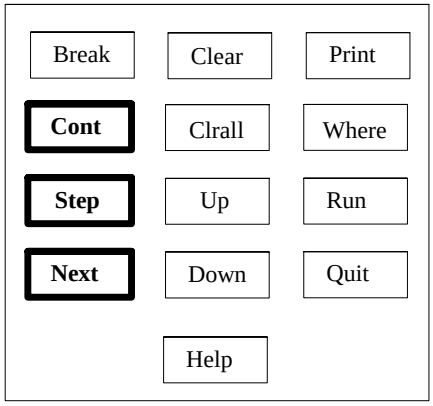
Displaying Variable Values



While macro execution is suspended, you can display the current values of any variables in a macro as follows:

1. Click on the variable name in the macro document to select it.
2. Click on the Print button in the debugger display. The value for the variable is displayed in the debugger window.

Stepping through Macro Execution



You can follow the execution of a macro from statement to statement by using the Step or Next buttons. You begin stepping through a macro at any break point.

To step through a macro, choose Step while at a break point. When you choose Step, the next executable statement in the macro is executed. You can then choose Step again to execute the next statement, or click the Cont button to continue processing of the macro without stopping after every executable statement.

If you choose Step while at a statement that calls another macro, the debugger steps into the new macro, and the Macro Editor display updates to show the new macro. You can then choose Step to step through the new macro's statements. When the new macro has been completely executed, the Macro Editor display returns calling macro.

You can also step through a macro using the Next button. Like Step, the Next button executes a single statement and stops. However, if you click Next while at a statement that calls another macro, the macro runs and processing continues with the next statement in the calling macro. Next does not step into the other macro.

Execution Stack

When you run a string of macros that call each other, ELF maintains an *execution stack.* An execution stack consists of the currently executing line of code, and all the lines of code that called macros before you

reached the currently running line of code. For example, consider this code sample:

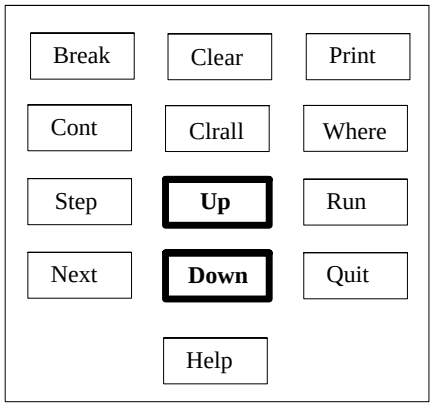
```
macro test
var x,string
string = "This is a test"
x = test1()                    ' This calls test1(). This line
info_message@(x)               ' is on the execution stack
endmacro                       ' while test1() is running.

macro test1
var y
y=test2()                      ' This line calls test2(). This
return(y)                      '  line is on the execution
endmacro                       ' stack while test2 is running.

macro test2
var z, a, b,c
z = 5
a = 1
b = 1
c = 1                                   ' BREAK POINT!
return(z)
endmacro
```

Suppose you set a break point at the line c=1 in test2(). This sequence occurs:

1. You run the macro test().
2. Test calls test1(). The line x=test1() is loaded onto the execution stack.
3. The macro test1() calls test2(). The line y=test2() is loaded onto the execution stack.
4. Execution stops at the line c=1. The execution stack now looks like this:

   x=test1()
   y=test2()
   c = 1

Moving Up and Down the Execution Stack



While you debug, it is often helpful to move up or down the execution stack in order to follow program execution or to set additional break points. For example, while you are following the execution of a sub-macro, you can move up the execution stack to the calling macro to set additional break points in the calling macro or to see the value of variables in the calling macro. You can then move back down the execution stack to the sub-macro to continue following its execution.

Moving Up the Execution Stack

To move up the execution stack, choose Up in the debugger display while macro execution is suspended. You can choose Up multiple times to progressively move up through the execution stack.

When you move up the execution stack, the Macro Editor display updates to show the calling macro.

NOTE: If the calling macro is an ELF system macro, the Macro Editor display does not show the calling macro. You can only display macros that you have written.

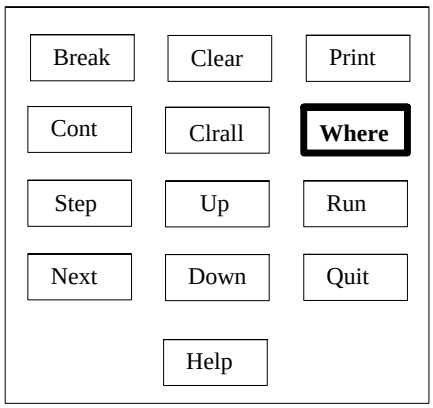
Moving Down the Execution Stack

To move down the execution stack, choose Down in the debugger display while macro execution is suspended. You can use the Down button only after you have moved up the execution stack using Up.

When you move down the execution stack, the Macro Editor displays updates to show the macro at the new execution level.

If you choose on Down while at the bottom of the execution stack, a message is displayed indicating that you can't move further down the stack.

Displaying Execution Stack Location



To see the current execution stack, click Where in the debugger display. Your current execution stack is displayed in the debugger window.

Debugging an Error Handler

ELF error handling is discussed in detail in Chapter 6. Error handlers are special routines that run when your program throws an error. Code Sample 5-1 contains a simple error handler.

```
Macro GetAge
 VAR str

again:
  ON ERROR
  {
        ERROR_BOX@
        GOTO again
  }
  str = PROMPT@("What is your age?")
  str = str + 0   'throws error if not a number
  IF str < 0 OR str > 120
        ERROR@(99,"Invalid age",str)
  RETURN(str)

ENDMACRO
```

Code Sample 5-1

When you run this program, you can force the program to throw an error 99 by typing in a number greater than 120 at the What is your age prompt.

The error handler in this program prints the error in a text box, then jumps to the again label. Not all error handlers are this simple, however, and you may occasionally want to debug your error handler if it is not working correctly.

The following procedure shows how to do this:

To set a break point in the error handler, and force the program to hit the breakpoint, follow these steps:

1. Copy and paste this program to the Macro Editor, compile it, and toggle Debugger Mode on.
2. Set your cursor at the beginning of the following line of code:

   ERROR_BOX@

3. Click **Break**. This sets a break point at the first line of the error handler.
4. Press **F8**.
5. Enter the name of the program to run in the dialog box. In this example, you would enter **getage**.
6. Click **OK**.

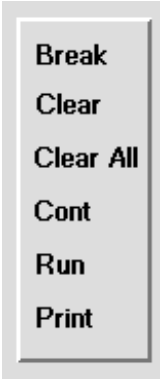
7. When the prompt appears, enter **200** in the entry box. This generates an error 99. The debugger reports the following error:

   error at line 14 in
   GETAGE:/newuser/applix/axhome/macros/macro1.am

8. Click **Cont**. The program hits the break point you set in the error handler. You can now step through the error handler a statement at a time, print the value of variables, and so on.

Right Mouse Button Options

Several debugger options are available from the right mouse button menu. Within the Debugger window, the following menu appears when you press the right mouse button:



These options correspond to the debugger buttons of the same name.

Debugger Messages

The debugger prints messages after each user action. This section lists the messages issued by the debugger, along with an explanation of each message. The messages listed are examples. Phrases or words that may vary are in italic.

Table 5-2 Debugger Error Messages

Error Messages	Description
*break at line 35 in /user/applix/axhome/macros/Macro3.am	This line appears when you successfully set a break point in your program.
*clear all breakpoints	This message appears when you click the clrall button
*clear break at line 34 in /user/applix/axhome/macros/Macro3.am	You successfully cleared a breakpoint.
*continuing...	This message appears when you click the Cont button. Execution continues until the next break point, or until the macro completes.
*could not break at line 34 in /user/applix/axhome/macros/Macro3.am	This message appears when your attempt to set a break point fails. The best way to clear the condition that prevents the break point from being set is: 1. Choose File → Compile. 2. Set the break point. 3. Rerun the macro.

Table 5-2 Debugger Error Messages (cont.)

Error Messages	Description
:down to TEST2: 31 in /user/applix/axhome/macros/Macro3.am	When you click the down button, you move to the next statement down the execution stack. This message tells you the macro name, the line number, and the macro document name that contains the next line of executable code on the execution stack.
no breakpoint at line *34* in */user/applix/axhome/macros/Macro3.am* to clear	You tried to clear a break point, but no break point was set at the line you selected.
No variable *'CALLED'* in macro *TEST*	You tried to print the name of a token, but that token is not a variable name.
*print *Y* in *TEST2*	You requested the debugger to print the value of the variable Y in the macro test2(). The value of the variable appears directly below this line.
*run *macroname*	This message appears when you click the run button. The line shows the name of the macro that you are running.
up to *TEST1*: *19* in */user/applix/axhome/macros/Macro3.am*	When you click the up button, you move to the next statement up the execution stack. This message tells you the macro name, the line number, and the macro document name that contains the next line of executable code on the execution stack.
TEST2:*34* in */user/applix/axhome/macros/Macro3.am*	This message appears when you press the where button. If there is more than one line on the execution stack, several lines like this appear.

6 Error Handling

One of the most elegant parts of the ELF language is its error handling. ELF supports thrown errors, which are errors that are passed up the execution stack for processing. This allows you to have a single error handler for a whole series of interconnected macros. This chapter covers the following topics:

- Introduction to Error Handling
- Error Handler Basics
- Layered Error Handlers

Introduction to Error Handling

If a macro generates an error condition, an information box describing the error is displayed and execution of the macro is terminated. For most purposes, allowing ELF to handle errors in this manner is sufficient. You can, however, include your own error handling procedures in macros. There are several reasons that you might want to write your own error handler:

- You have developed an application that produces unique error situations. You can include procedures in your application to handle the errors appropriately.
- The macro takes a significant amount of time to process. You might include error handling procedures to test for common errors so that execution of the macro is quickly terminated if an error condition exits.
- You might not want your macro to terminate when errors are encountered. For instance, you might want to respond to a particular type of error by re-displaying a dialog box so that the user can supply different information.

The Components of an Error

When an error is encountered by ELF, an error code is thrown. The error that is thrown can be an ELF built-in error, or an error you define using the ERROR@ macro. A list of common ELF built-in errors is contained in the file error_.am.

An ELF error consists of as many as three elements:

- The error code is a unique error code number. For ELF built-in errors, the error numbers have been mapped to descriptive words

using DEFINE statements. For example, the error code 4116 is defined as ERR#NOT_AN_ARRAY_ in the file errors_am.

When you create your own errors, the error code must be a number in the range of 1 to 1000.

- The error string is descriptive text that is displayed in the error message box when the error is triggered.
- The error object is a string that represents the item to which the error applies. For example, if an error is generated because a file cannot be opened for writing, the error object could be the name of that file.

  Including an error object with an error is optional. If an error object is present, it is displayed preceding the error string in an error dialog box.

The following section describes the structure and the procedures for creating and throwing errors, and for catching errors using error handlers.

Error Handler Basics

Unless you provide an error handler to catch an error, ELF handles the error by displaying an error message box and terminating the macro that produced the error.

If you provide an error handler, the following sequence occurs:

1. The error handler processes the error, according to the code you provide.

2. The error handler is disabled. There is an implied ERROR ON ERROR statement at the end of every ELF error handler. *Implied* means it is not visible to the programmer, but it executes anyway.

3. The ELF macro proceeds by executing the same line of code that produced the error. Typically, you use a GOTO somewhere in the error handler to direct the macro to a label. If you omit the GOTO statement from your error handler, ELF restarts execution with the same line that caused the error condition.

Structure of an Error Handler

Every error handler starts with the ON ERROR statement. The format of an ON ERROR statement is:

ON ERROR *statement*

Where *statement* is any ELF statement. Error handlers can be only one line. For example:

```
ON ERROR INFO_MESSAGE@("You sunk my battleship!")
```

An error handler can contain many lines. You can use curly braces for multiple-line routines. For example:

```
ON ERROR
{
  IF ERROR_NUMBER@() = 45
  {
      ERROR_BOX@
  }
 ELSE RETHROW_ERROR@
}
```

Activating and Deactivating Error Handlers

A macro can contain any number of error handlers, but only one error handler can be active at any time. A handler becomes active when the

ON ERROR statement is encountered during processing. The handler remains active until one of the following occurs:

- The handler catches an error. Once an error is caught, the handler is automatically deactivated and remains so unless it is executed again.
- Another ON ERROR statement is encountered. This new ON ERROR statement becomes the current error handler.
- An ERROR ON ERROR statement is encountered. You can use the ERROR ON ERROR statement to disable the current error handler. When the current error handler is disabled, errors are handled by the next error handler up the chain.

Example: Activating and Deactivating an Error Handler

Figure 6-1 shows the macro MACRO2. This macro contains an error handler that displays the current error, then uses a GOTO to resume execution with the again label. This label is located at the top of the macro so that the error handler is re-enabled after it executes.

MACRO2 has an ERROR ON ERROR statement in it that is commented out. If you remove the comment mark (') and recompile MACRO2, MACRO2 executes the ERROR ON ERROR statement and its error handler is disabled. Any error thrown by MACRO2 after its error handler is disabled is caught by the ELF error handler.

ELF Code	Comments
macro Macro2 VAR str again: ON ERROR { ERROR_BOX@ GOTO again } str = PROMPT@("What is your age?") str = str + 0 'throws error if not a number ' ERROR ON ERROR IF str < 0 OR str > 120 ERROR@(99,"Invalid age",str) RETURN(str) ENDMACRO	Try copying this code to the Macro Editor, then compile and run it. This is a good way to familiarize yourself with error handling. When the box prompts you for an age, enter a number greater than 120. This generates an error 99. The ERROR ON ERROR statement in MACRO2 disables the error handler in MACRO2. This means that if you enter a number greater than 120 or less than 0 in the entry area, The ELF error handler catches the error. Every error handler has an implied ERROR ON ERROR statement in it which disables the error handler after it catches an error. When your error handlers are catching multiple errors, you should be sure you know which error handler is enabled when the second error occurs.

Figure 6-1 Activating and Deactivating an Error Handler

Types of Error Handlers

There are several types of error handlers. They can be divided into four types:

- General purpose error handler. This type of handler catches all errors and displays them to the screen.

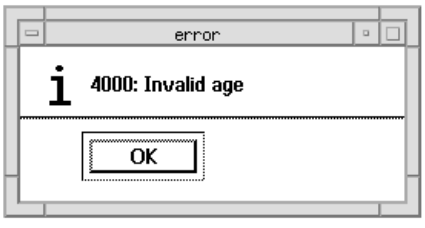
- Logger. This type of error handler catches errors, and writes the error information to a disk file. This is very convenient for the system administrator who wants to look at the error at some later time.
- Error handlers that catch a particular error and perform operations based on the error. For example, you may have errors that indicate a user has typed something wrong and needs to retype the item. Rather than terminating the macro, you might want to display the error and then allow the user to type the item again.
- Error handlers that catch all errors except a particular error or set of errors. You might have a macro for which you want to handle most of the possible errors. You can create a handler that indicates what errors to handle and what errors to ignore. Errors that are ignored can be rethrown so that another error handler can catch them.

Displaying an Error Dialog Box

When you catch an error using an error handler, you may want to display the error. Use the ERROR_BOX@ macro to do this:

```
ERROR_BOX@()
```

The string and object from the most recently thrown error are used in the error message box.



Rethrowing an Error

Typically, error handlers in ELF macros are written for a small number of commonly-encountered errors. Other errors are rethrown to a general-purpose error handler, or to ELF. *Rethrowing an error* means passing the error to the next error handler up the execution chain.

To rethrow an error, you use the ERROR_RETHROW@ macro:

```
ERROR_RETHROW@()
```

ERROR_RETHROW@ rethrows the most recently thrown error. If no error has been thrown, ERROR_RETHROW@ throws the error ERR#ILLVAL_. This constant is defined in the ELF include file errors_.am.

Error Handler Examples

This section presents examples of error handlers that are used to perform a number of different operations. Each example is based on a function that prompts for an age value and then evaluates whether the given age is valid.

Catching all Errors

This example includes a simple error handler that catches all errors. When an error is caught, ERROR_BOX@ is used to display the error dialog box and processing of the macro continues at the again label. This error handler would be useful when you want the operation of a macro to continue when an error is encountered rather than have the macro terminate.

NOTE: The again label is at the top of the macro. When you hit an error, the error handler processes the error, then terminates. By

processing the ON ERROR statement after an error, the error handler is re-established. If you do not re-establish the error handler after the first error is processed, the second error is handled by the ELF error handler.

```
FUNCTION GetAge
  VAR str
again:
  ON ERROR
  {
       ERROR_BOX@
                  GOTO again
  }
  str = PROMPT@("What is your age?")
  str = str + 0              'throws error if not a number
  IF str < 0 OR str > 120
       ERROR@(99,"Invalid age",str)
  RETURN(str)
ENDMACRO
```

Catching a Specific Error

The error handler below processes the error produced if you type an invalid age. If any other errors are thrown, the ELF error handler catches it and terminates the GetAge macro.

```
FUNCTION GetAge
  VAR str

again:
  ON ERROR
  {
       IF ERROR_NUMBER@() = 99
       {
         ERROR_BOX@
         GOTO again
       }
}

  str = PROMPT@("What is your age?")
  str = str + 0                              'Converts string to a number
  IF str < 0 OR str > 120
       ERROR@(99,"Invalid age",str)
  RETURN(str)
ENDMACRO
```

Ignoring Specific Errors

In this example, all errors except the error produced by an invalid age being typed are handled by the error handler. If the invalid age error is thrown, the handler rethrows the error so that it will be handled by the ELF error handler.

```
FUNCTION GetAge
  VAR str

again:
  ON ERROR
      {
      IF ERROR_NUMBER@()<> 99
                GOTO again
      ELSE
                ERROR_RETHROW@
      }

  str = PROMPT@("What is your age?")
  str = str + 0
  IF str < 0 OR str > 120
      ERROR@(99,"Invalid age",str)
RETURN(str)

ENDMACRO
```

Layered Error Handling

When you have several interconnected ELF macros that have their own error handlers enabled, how an error is handled depends on where the error is caught. There are three possibilities:

- An error handler for the generated error is included in the macro or function that produced the error.

- An error handler for the error is present in one of the macros up the execution stack from the macro that produced the error.
- If no handler for the error is present in the macros, the error is handled by ELF. ELF handles an error by displaying an error message box and then terminating the macro that produced the error.

Example: Layered Error Handling

Figure 6-2 shows two macros: MACRO1 calls MACRO2. When MACRO2 executes the ERROR ON ERROR statement, its error handler is disabled. Any error thrown by MACRO2 after its error handler is disabled is caught by the MACRO1 error handler.

If you remove the ERROR ON ERROR statement from MACRO2, the first error is handled by the MACRO2 error handler. The second error is handled by the MACRO1 error handler. The third error is handled by the MACRO2 error handler, because after the MACRO1 error handler runs, it runs MACRO2 a second time, re-establishing the MACRO2 error handler. All other errors are handled by ELF.

ELF Code

```
macro Macro1
var x
ON ERROR
{
  Info_Message@("You have an Error")
}
x = Macro2()
ENDMACRO


macro Macro2
VAR str
ON ERROR
{
  ERROR_BOX@
}
str = PROMPT@("What is your age?")
str = str + 0     'throws error if not a number
ERROR ON ERROR

IF str < 0 OR str > 120
  ERROR@(99,"Invalid age",str)
RETURN(str)

ENDMACRO
```

Comments

Try copying this code to the Macro Editor, then compile and run it. This is a good way to familiarize yourself with layered error handling.

When the box prompts you for an age, enter a number greater than 120. This generates an error 99.

The ERROR ON ERROR statement in MACRO2 disables the error handler in MACRO2. This means that if you enter a number greater than 120 or less than 0 in the entry area, The error handler in MACRO1 handles the error. If you enter an invalid value a second time, the ELF error handler catches the error.

Every error handler has an implied ERROR ON ERROR statement in it which disables the error handler after it catches an error. When your error handlers are catching multiple errors, you should be sure you know which error handler is enabled when the second error occurs.

Figure 6-2 Error Handling in Nested ELF Macros

Logging

Logging is an important type of error handling, particularly for applications that are large and widely used. A logger writes error information to a file, rather displaying it on the screen. This is important if you are trying to track problems in a large application that might be generating errors in any number of macros.

The code below shows a simple logger. Errors are written to /tmp/macro.log.

```
macro Logger
VAR str, error_array
ON ERROR
{
error_array = "Error File:  "++ERROR_FILE@(),
                    "Line Number:  "++ERROR_LINE@(),
                    "Macro Name:  "++ERROR_FUNCTION@(),
                    "Error Number:  "++ERROR_NUMBER@(),
                    "Error Message:  "++ERROR_STRING@()
WRITE_ASCII_FILE@("/tmp/macro.log", error_array)
}
str = PROMPT@("What is your age?")
str = str + 0     'throws error if not a number
IF str < 0 OR str > 120
  ERROR@(99,"Invalid age",str)

ENDMACRO
```

Error Macros

There are six built-in macros for returning information on the most recently thrown error:

- ERROR_FILE@ returns the name of the source file containing the macro that caused the last error to occur.
- ERROR_LINE@ returns the line number of the most recently thrown error.
- ERROR_FUNCTION@ returns the name of the last macro to throw an error.
- ERROR_NUMBER@ returns the error code for the most recently thrown error.
- ERROR_OBJECT@ returns the error object for the most recently thrown error. If the most recently thrown error does not include an object, this macro returns NULL.
- ERROR_STRING@ returns the error string from the most recently thrown error.

ERROR@

You can define your own error messages using the ERROR@ macro. The format for the macro is:

```
ERROR@(code,string[,object])
```

The *code* is a unique error number. The number must be in the range of 1 to 1000. The *string* is the error string that is displayed to the user. The *object* is a string representing the object of the error. Including an error object is optional.

For example, the following creates an error identified by error number 100. No error object is used.

```
ERROR@(100, "Name not recognized")
```

The ERROR@ macro throws an error. For example, the following function is used for getting an age value from a user. The function checks whether the age typed is valid. If it isn't, the error defined by ERROR@ is thrown.

```
FUNCTION GetAge
    VAR str
    str = PROMPT@("What is your age?")
    IF str < 0 OR str > 120 THEN
            ERROR@(99, "Invalid age", str)
    RETURN(str)
ENDMACRO
```

This example uses an error object. When the error is thrown, the string that was typed as the age is passed as the error object with the error. When the error is caught and displayed (either by an error handler you create or by the ELF error handler), the error message box includes the age object.

7 Dialog Box Editor

The primary interface between a user and your ELF macro is typically a dialog box. A dialog box is a collection of programmable, graphical user interface controls that present information to the user, and pass user events to ELF. This chapter discusses the following topics:

- Introduction to the Dialog Box Editor
- Building a dialog box
- Other dialog box features

Introduction to the Dialog Box Editor

The Dialog Box Editor allows you to create the layout for dialog boxes. Using this tool, you specify the size of the dialog box and the type and placement of controls within the dialog box.

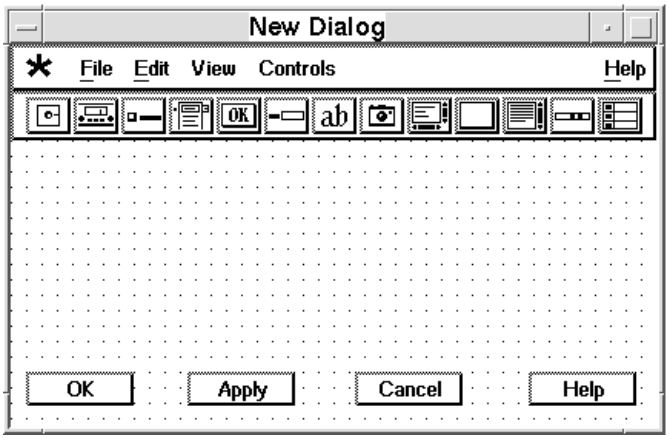
When you are satisfied with the design of your dialog box, exit the editor and the dialog box is saved in a file. It is important to note that the Dialog Box Editor is used to design the layout of a dialog box, not to define its functionality. Once you have designed the dialog box, you include statements in a macro document to display the dialog box and perform operations based on interaction bvetween the dialog box and the user.

Starting the Dialog Box Editor

You can use the Dialog Box Editor to create a new dialog box, or edit an existing dialog box. The Dialog Box Editor is started from within the Macro editor.

To use the Dialog Box Editor to edit a new dialog box, follow these steps:

1. Run the Macro Editor.
2. Choose **Tools → Create Dialog Box.** The Dialog Box Editor is displayed.

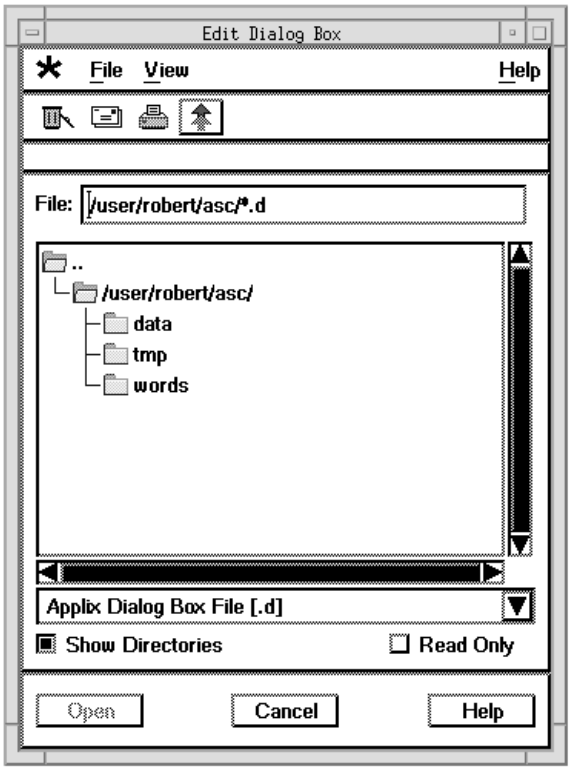


A new dialog box containing four buttons is displayed. The buttons are labeled OK, Apply, Cancel, and Help.

To use the Dialog Box Editor to edit an existing dialog box, follow these steps:

1. Run the Macro Editor.
2. Choose **Tools → Edit Dialog Box**.

   The Edit Dialog Box dialog is displayed showing a list of all the dialog box files (files having the .d extension) in your macros directory.

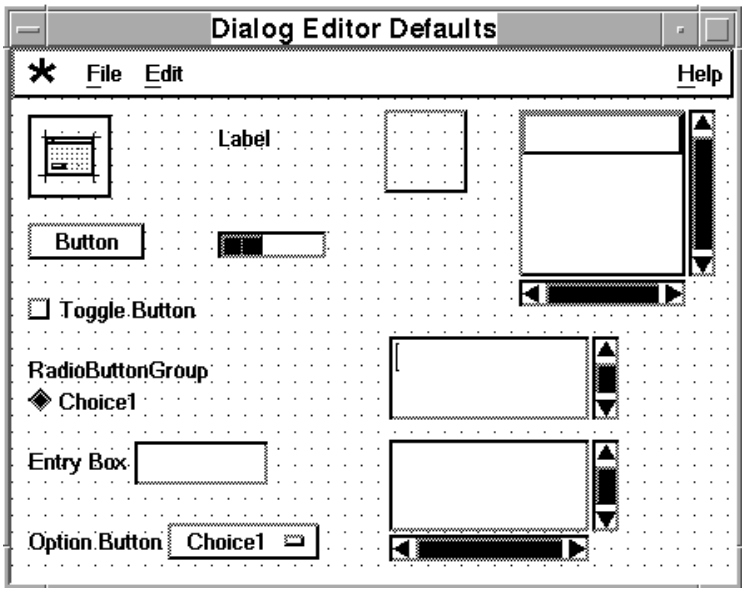


If the dialog box file you want to edit is not in your macros directory, specify the directory you want to display in the File entry field.

3. Double-click on the name of the dialog box file you want to edit. The Dialog Box Editor displays and the dialog box you specified is loaded.

Changing Default Control Settings

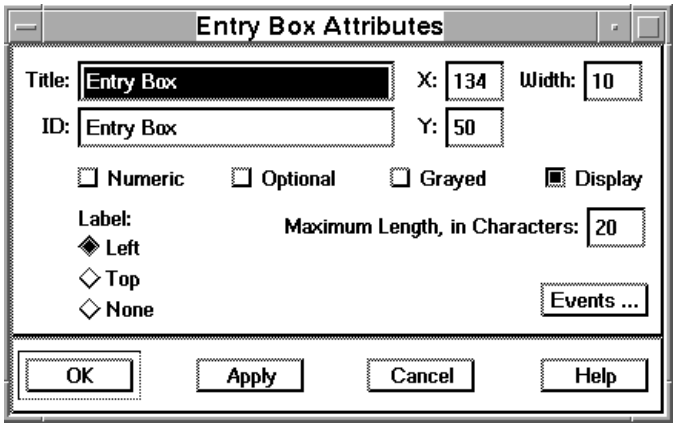
You can change the default settings for any or all of the dialog box controls with the Dialog Editor Defaults dialog box. Choose Edit → Dialog Defaults and the Dialog Editor Defaults dialog box appears.



The default for each ELF control appears in the dialog box. When you add a control to a dialog box, the initial settings of that control are identical to the settings established in this screen. Once a control is added to your dialog box, you can change the attributes of that instance of the control. This is described in the section, "Changing Dialog Box Settings," later in this chapter.

You can change the default attributes, such as size, title, or ID, for any of the controls. Double-click on a control and the dialog box for that control appears.

For example, if you double-click on the entry box control, the Entry Box Attributes dialog box appears.



See the following section, "Adding and Positioning Controls," and Chapter 10, "ELF Control Reference" for information about control attributes.

Configuring Dialog Box Colors

To use colors in your dialog box, you must configure Applixware as follows:

Table 7-1 Applixware Color Configuration Parameters

Preference or Menu Sequence	Setting
Color → Standard Color Map field	NULL or XA_RGB_APPLIX_MAP
Color → Force Color Widgets to Display Monochrome toggle	OFF
Color → Force All Text Widgets to Use Work Area Background Color	Optional - can be on or off. See the discussion later in this section.

Table 7-1 Applixware Color Configuration Parameters (cont.)

Preference or Menu Sequence	Setting
Color → Number of Colors to use for applications	8 or greater.
Color → Applixware Display Colors	Set as desired. See the section "Force All Text Widgets to Use Work Area Background Color".

Once you set these configuration parameters, shut down and restart Applixware so that the changes take affect. With these configuration parameters set, you can change the colors of your dialog boxes, and the controls in those dialog boxes. A description of these parameters is available from the on-line help topic Color Preferences. The settings indicated in Table 7-1 are described in more detail in the following sections.

Standard Color Map Field

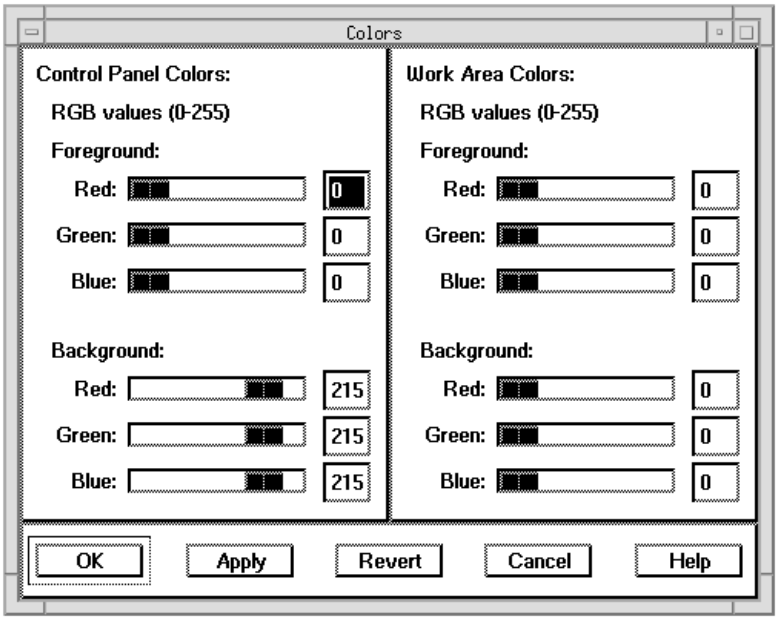
You must load the Applixware default color map before you can manipulate colors. Leave the field blank, or enter the string XA_RGB_APPLIX_MAP in the Standard Color Map field.

Force Widgets to Display Monochrome

This toggle button must be turned off. Monochrome monitors do not support colors.

Force All Text Widgets to Use Work Area Background Color

Colors for ELF controls (widgets) and dialog boxes are set through the Color Preferences dialog box under ✶ → Applix Preferences → Color → Applixware Display Colors.



If Force All Text Widgets to Use Work Area Background Color is turned on, the color of your dialog box is controlled by the Work Area Colors: Background sliders, and the color of the text of your controls is controlled by the Work Area Colors: Foreground sliders.

If Force All Text Widgets to Use Work Area Background Color is turned off, the color of your dialog box is controlled by the Control Panel Colors: Background sliders, and the color of the text of your controls is controlled by the Control Panel Colors: Foreground sliders.

Number of Colors to Use for Applications

You should set this setting to a number higher than 2. Setting the value to 2 limits the choices you have for custom control colors to black or white. The available choices are 2, 8, 27, 64, 125, and 216.

Building a Dialog Box

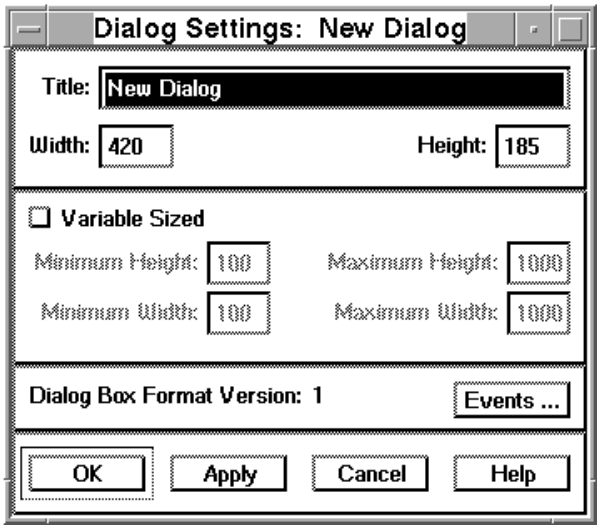
Once your Dialog Box Editor is configured, you are ready to build a dialog box. Building a dialog box requires the following five steps:

1. Changing dialog box settings.
2. Adding and positioning controls.
3. Configuring color.
4. Choosing character settings.
5. Saving and exiting the Dialog Box Editor.

Changing Dialog Box Settings

After you open a dialog box, you can specify the dialog settings, including the dialog box title, size, and optional event functions.

To change the dialog box settings after opening a dialog box, choose Edit → Dialog Settings. The Dialog Settings dialog box appears.



This dialog allows you to change the following dialog box characteristics:

Title	This is the string displayed at the top of the dialog box.
Width	The width of the dialog box when it is first displayed.
Height	The height of the dialog box when it is first displayed.
Variable Sized	Determines whether you can change the size of the dialog box.
Events	There are four dialog box events: Dialog Initialization, Poke, Resize, and Menu Bar events.

These settings are described in more detail in the sections that follow.

Title

The Title entry area shows a string that is displayed at the top of the dialog box when you run your macro. When you create a new dialog box, the name New Dialog is placed in the entry area. To change the title of the dialog box, follow these steps:

1. Double-click in the Title entry area.
2. Type the new name.

Width and Height

The initial width and height of the dialog box appear in the Width and Height entry areas. To change the width or height of the dialog box:

1. Double-click in the Width or Height entry area.
2. Type the new dimensions.

You can also change the Width and Height values by resizing the dialog box with your mouse. To do this,

1. Click on the corner of the dialog box in the dialog editor
2. Drag to establish a new size for the dialog box. The new width and height settings appear in the Dialog Settings dialog.

Variable Sized

A dialog box cannot be resized unless you turn the Variable Sized toggle on. Enter the minimum and maximum width and height in the appropriate entry boxes. These settings establish the maximum and minimum size of the dialog box in the Dialog Box Editor, and at run time.

Events

Click Events to display the Dialog Events Functions dialog box, which contains entry areas for ELF macros for the following optional dialog event functions:

Dialog Initialization
: The Dialog Initialization macro is called before the dialog box appears and initializes the dialog box attributes. The dialog box handle returned by DB_LOAD@ is passed as a single argument to the function.

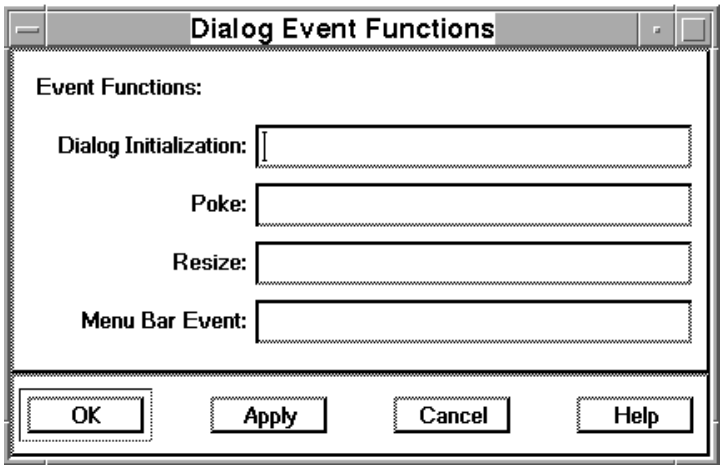
Poke
: The Poke entry area is for an ELF macro that executes when the dialog box receives any poke message, defined by DB_ACCEPT_POKES@, acceptable to the macro. A poke event function is called. The dialog box handle returned by DB_LOAD@ is passed as a single argument to the function.

Resize
: The Resize entry area is for an ELF macro that executes when the dialog box resizes. The dialog box handle returned by DB_LOAD@ is passed as a single argument to the function.

Menu Bar Event
: The Menu Bar Event is for an ELF macro that executes when a dialog box menu bar control is selected. The dialog box handle returned by DB_LOAD@ is passed as a single argument to the function.

You can type the name of a macro in the appropriate entry area. These events apply to only the current dialog box. They affect every poke event, resize event and menu bar event that occurs within any macro that loads the current dialog box. The initialization event triggers

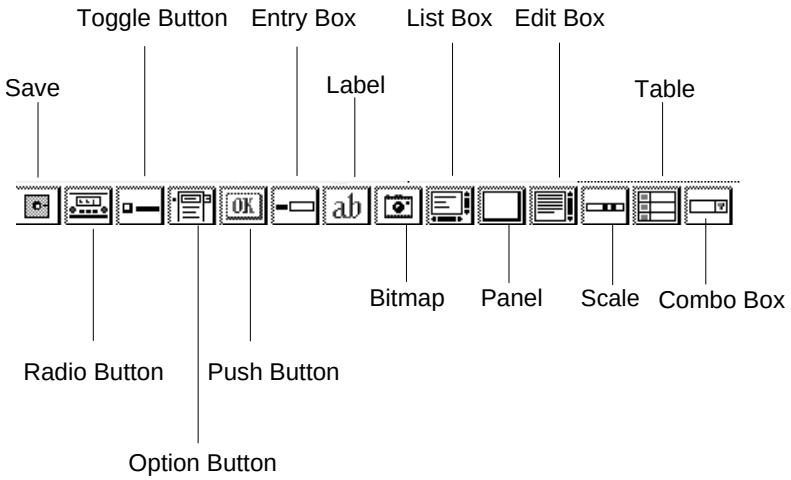
whenever the current dialog box is initialized with the DB_LOAD@ macro.



Adding and Positioning Controls

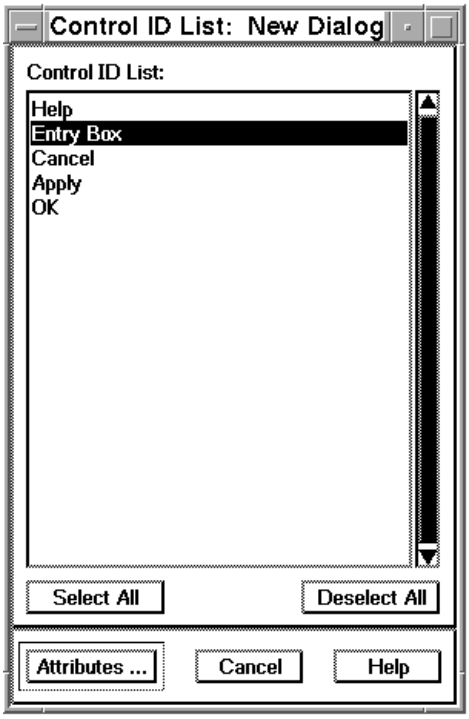
To add a control, you can either:

- Choose a control from the Controls menu, then click anywhere in the dialog box.
- Click on an *Express*Line control icon, and click anywhere in the dialog box.



Selecting Controls

You can select a control by clicking on the control in the dialog box. You can also choose Edit → Select to display the Control ID list dialog box.



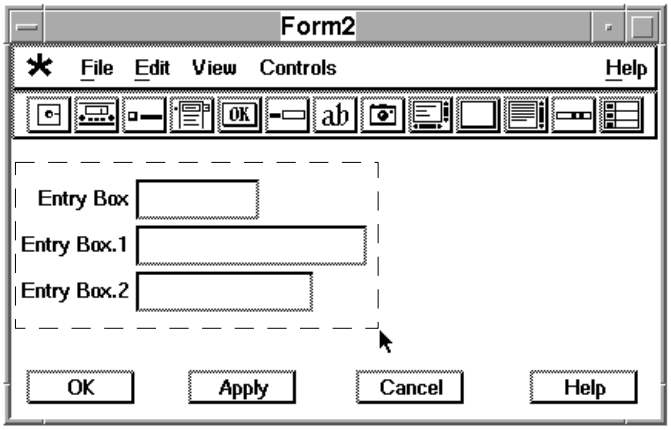
To select a control, you can click on a control ID in the Control ID List area, or you can click Select All for all controls in the dialog box. Click Attributes to change the attributes of selected controls.

If you have hidden controls in a dialog box, which have their Display toggle button turned off, you can use the Control ID list dialog to select those controls and manipulate their attributes.

Selecting Multiple Controls

You can select multiple controls in a dialog box using one of the following methods.

- Press and hold down the mouse button and drag the mouse pointer. A selection rectangle appears as you drag the mouse pointer as in the following example:



  All objects enclosed within the selection rectangle are selected when you release the mouse button.

- Select each item individually. Select the first object by clicking on the border of the object. Select additional objects by holding down the CTRL key while clicking on the additional objects. Holding down the CTRL key enables you to select a new object without de-selecting other objects.

You can combine the two selection methods. For example, you can use a selection rectangle to select objects located near each other, then use CTRL-click to select another object that is apart from the other objects.

Positioning Controls

You can position controls in the dialog box with your mouse. Click on the control and drag it to its new position.

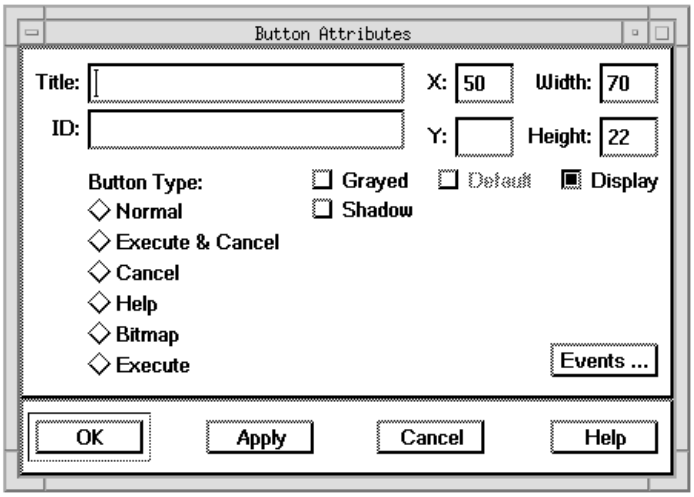
To align multiple controls, you can use the X and Y attributes settings for each control. If you set the X value of each control to the same value, the left side of each control is aligned.

For example, suppose you have four buttons in a dialog box, and you want the buttons to be aligned in a column. Follow these steps:

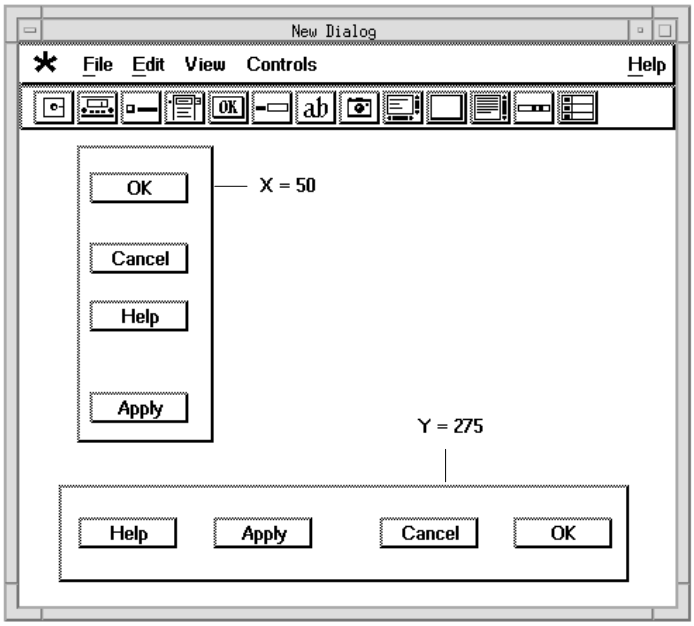
1. From the Dialog Box Editor, choose **Edit → Select**. The Control ID List appears:

2. Select the buttons that you want to align by pressing Shift, and clicking the name of the button. You can select one or several.

3. Click Attributes. The Button Attributes dialog appears.

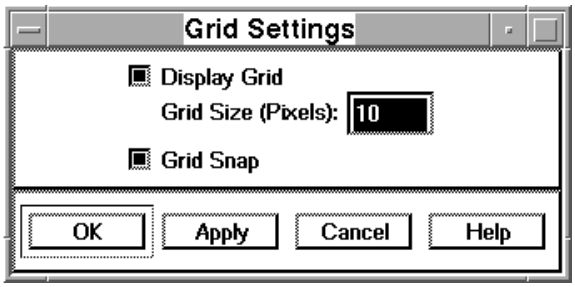


4. Enter a value in the X field to align the buttons in a column. Enter a value in the Y field to align the buttons in a row. The following illustrates these results.



Grids

You can also use grids to position the controls in the dialog box. To configure grids in the Dialog Box Editor, choose View → Grids. The Grid Settings dialog box appears.



You can turn on the grid, choose the grid size, and turn on Grid Snap. Grid Snap automatically positions the left corner of controls on the grids.

Borders

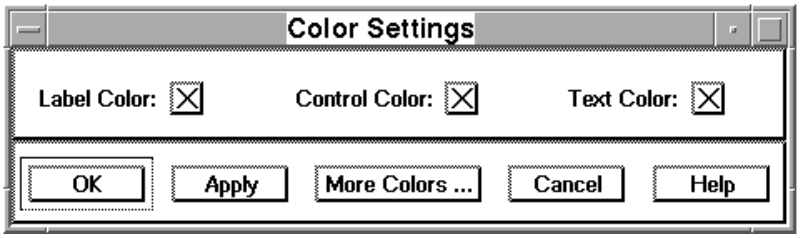
Choose View → Borders to turn on borders around dialog box controls. Borders are only visible in the Dialog Box Editor. They are not part of the dialog box. Use borders to show the mouse-sensitive area of dialog box controls.

Setting Color in a Dialog Box

Controls appear in a dialog box with the default color settings. You can change the color settings for controls, control labels, or text in a dialog box with the Color Settings dialog box. Colors can be set individually for each control, allowing you to highlight controls in a dialog box.

To change a control's color settings:

1. Choose the control in the dialog box.
2. Choose **Edit → Color Settings**.



3. Change the color settings in the Color Settings dialog box.

Control, label, and text color are initially set to the default color. Choose **Edit → Color Settings** and click on a **Color** option to change your color preference setting. When you click on a color option, the available colors in the color pallete appear. You can choose a color from the color pallete, or create your own color with the Edit Color Pallete dialog box.

The Edit Color Pallete dialog box allows you to add or delete colors from the current color pallete. To change the current color palette from the Color Settings dialog box:

5. Click on **More Colors**.
6. Change the color pallete.

   The Color area shows the available colors you can add to the palette. You can click in the Color area to choose a color, or you can use the HSB scale options to adjust the hue, saturation, and brightness of the color. The HSB settings for the current color in the Color area appear beneath the scales, and the current color appears in the Current Color area. If you want to add a color to the current color palette:

7. Click on a color in the Color area or use the HSB scale options to set a color.
8. Type a name in the Color Name entry area.
9. Click **Add**.

   The color is added to the current color palette and is immediately available in the Color Settings dialog box for control, label, and text colors.

If you want to remove a color from the current color palette:

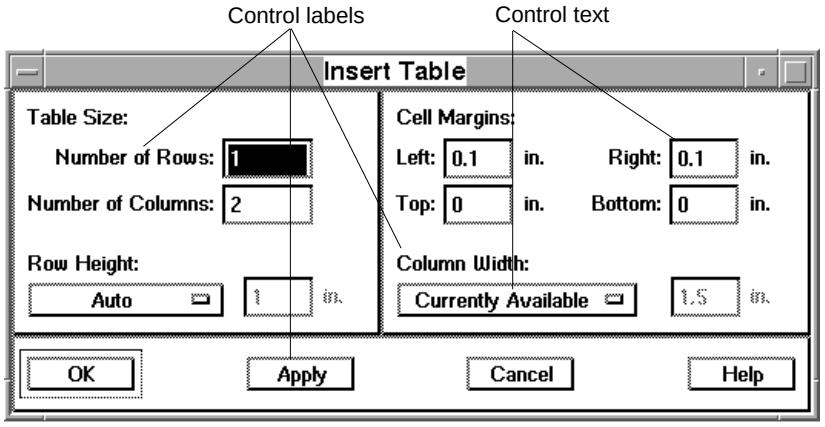
1. Choose the color with the **Current Palette** option.
2. Click **Delete**.

You can save a color palette as a file, allowing you to use the same color palette in different dialog boxes. After you have saved color palettes, you can choose File → Open to load a color palette in the current dialog box.

Choosing Character Settings

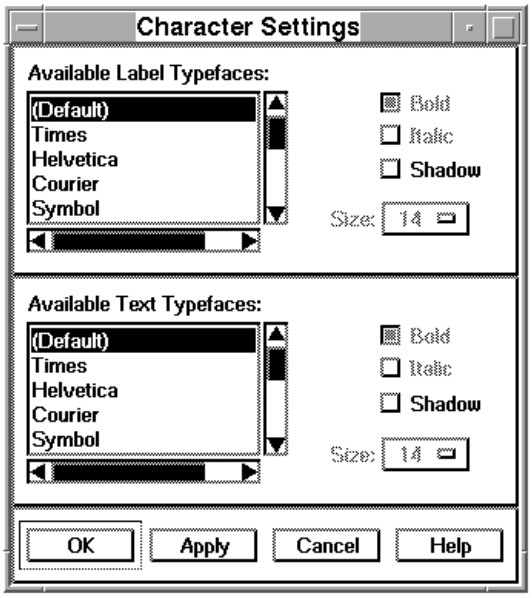
Controls appear in a dialog box with the default label and text settings. You can change the character settings for control labels or text in a dialog box with the Character Settings dialog box.

A label is the text that appears as the title of a control, for example, an entry box label or a push button name. Text is the text that appears in the control, for example, the text typed in an entry area or the choices in an option button.



To change a control's character settings:

1. Choose the control in the dialog box.
2. Choose **Edit → Character Settings**.



3. Change the label and text character settings in the Character Settings dialog box.

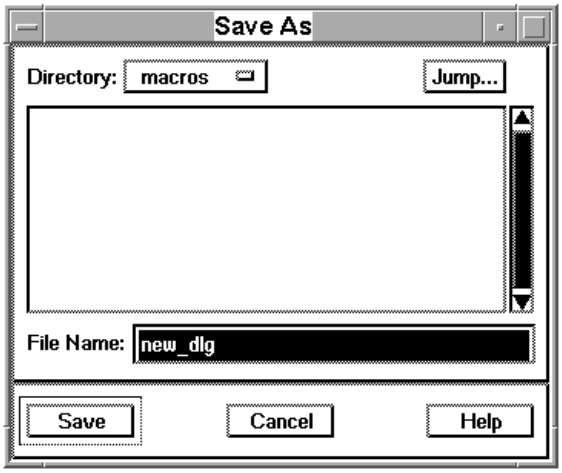
The options in the Character Settings dialog box are grayed if the option is unavailable. For example, a push button has a label, but it does not have text. When you choose a push button in a dialog box, the Text options in the Character Settings dialog box are grayed.

Saving and Exiting the Dialog Box Editor

When you exit the Dialog Box Editor, the Save Dialog Edits dialog box appears.

You can click on Save to accept all the changes you've made to the dialog box, or you can click on Discard to exit without saving any of the changes.

If you click on Save and the dialog box is a new one that has never been saved, the Save As dialog box is displayed.



The File Name entry area displays a file named new_dlg in your current directory. The file is automatically saved with the .d file name extension, indicating that the file is a dialog box definition file. Type a new name in place of the new_dlg file name. Click Save to save the dialog box file.

Other Dialog Box Editor Features

The Dialog Box Editor allows you to perform several basic editing functions on controls in your dialog box. This section described the following functions:

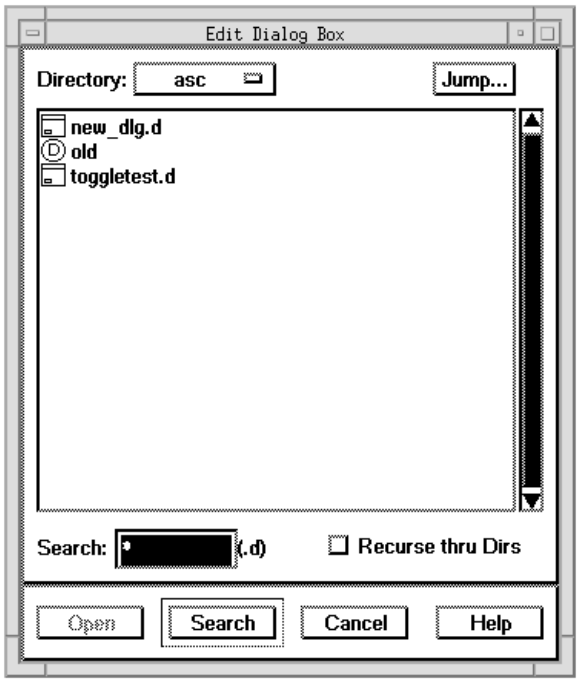
- Editing an Existing Dialog Box
- Deleting Controls

- Cut, Copy, Paste, and Paste Dialog
- Using Undo
- Adding Accelerator Keys

Editing an Existing Dialog Box

To edit an existing dialog box, follow these steps:

1. From the Macro Editor menu, choose **Tools → Edit Dialog Box**. The Edit Dialog Box dialog appears.



This dialog shows only dialog box files, which have a .d extension.

2. Use the directory pulldown and jump button to locate the directory containing the dialog box you want to edit.
3. Double-click the dialog box file to open it in the Dialog Box Editor.

Deleting Controls

To delete a control from the dialog box:

1. Click on the control in the dialog box.
2. Choose **Edit → Delete Selected**.

You can delete more than one dialog box control at one time with the Control ID list dialog box. To delete multiple controls at one time:

1. Choose **Edit → Select**.
2. Choose the controls in the Control ID List dialog box.

   The controls appear selected in the dialog box.
3. Choose **Edit → Delete** Selected.

Note that once you delete a control, it is not recoverable.

Cut, Copy, Paste, and Paste Dialog

The Dialog Box Editor contains cut, copy, and paste features to handle control editing.

To remove selected controls from the dialog box and move them to the clipboard, choose Edit → Cut. To put a copy of selected controls onto the clipboard, choose Edit → Copy. Material remains on the clipboard until you overwrite it by cutting or copying other material.

Use Edit → Paste to put material from the clipboard into the dialog box. You can repeatedly paste the same material, since the clipboard remains intact until more material is cut or copied onto it.

To activate this option, you must select at least one control.

Control IDs

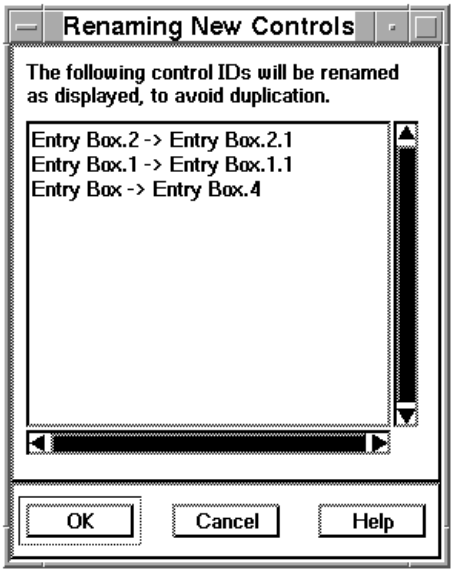
When you create or copy a control, a *control ID* is assigned to the control. The ID is a string by which the control can be referenced within an ELF program.

The Control ID is used by the DB_CTRL family of macros, that allow you to manipulate dialog box controls within your ELF macro. For example, the macro DB_CTRL_DISPLAY@ allows you to enable and disable a dialog box control.

You can change the control ID in the control's attributes dialog box by typing a new string in the ID: field. This ID must be unique within the dialog box.

Avoiding Duplicate Control IDs

The Dialog Box Editor contains safeguards to avoid the duplication of control IDs. When you copy a dialog box control with Edit → Copy, then choose Edit → Paste, the Renaming New Controls dialog box appears.



The Renaming New Controls dialog box also appears when you choose Edit → Paste Dialog, and at least one of the controls in the pasted dialog box has the same name as a control in the current dialog box.

The original control ID appears in the list box with the proposed renamed control ID. Click on OK to accept the proposed renaming of the controls, or click on Cancel to abandon the paste.

To paste the contents of an existing dialog box file into the current dialog box choose Edit → Paste Dialog. The Paste Dialog Box dialog appears.

The dialog box shows all the dialog box files and directories in the current directory. Double-click on the document you want to open, or change directories to a different dialog box.

Undo

The Dialog Box Editor window has a one-level undo option to cancel the most recent edit action. To cancel an edit action, choose Edit → Undo.

Adding Accelerator Keys

You can assign accelerator keys to ELF dialog box controls through the dialog box editor. To do this, follow these steps:

1. Run the Macro Editor.
2. Choose **Tools → Create Dialog Box**.
3. Add some controls to the dialog box.
4. Choose **Edit → Accelerator Keys**. This dialog box appears:

5. Enter a key for the controls you want to be able to address through keystrokes. For example, if you enter T for a toggle button, you will be able to shift focus to turn the toggle button on and off by pressing <ALT>-<T>.

Assigning keystrokes to some widgets works in slightly different ways:

- Radio Button Group - The individual items in the radio button group are assigned accelerators through the Radio Button Attributes dialog box. Double-click on a radio button group to access this dialog box.

- Edit boxes, List boxes, Scale widgets - None of these widgets have visible titles. They can still be assigned accelerator keys. Position a label near these controls to provide a title. Give the same mnemonic to the label and Edit Box, List Box, or Scale widget. The

dialog box code in Applixware knows a label cannot be selected, so transfers focus to the nearest control with the same mnemonic.

If you want to make changes to the dialog boxes in Applixware, the product dialog boxes are in /*install_dir*/axdata/eng/dialogs. These can be edited in place. The edits take effect the next time the dialog box is called by an application.

8 ELF Control Reference

This chapter describes the controls available with the Dialog Box Editor. For each control, there is a discussion of the attributes and events associated with the control. At the end of each discussion is a code sample that uses the control and shows how it works.

The following controls are described:

Bitmap, Combo Box, Edit box, Entry Box, Label, List Box, Option button, Panel, Push button, Radio Button Group, Scale, Table, Toggle Button

Introduction to ELF Controls

There are twelve ELF controls available from the dialog box. The control described in this chapter are listed in Table 8-1.

Table 8-1 ELF Dialog Box Controls

Bitmap	Combo Box	Edit Box
Entry Box	Label	List Box
Option Button	Panel	Push Button
Radio Button Group	Scale	Table
Toggle Button		

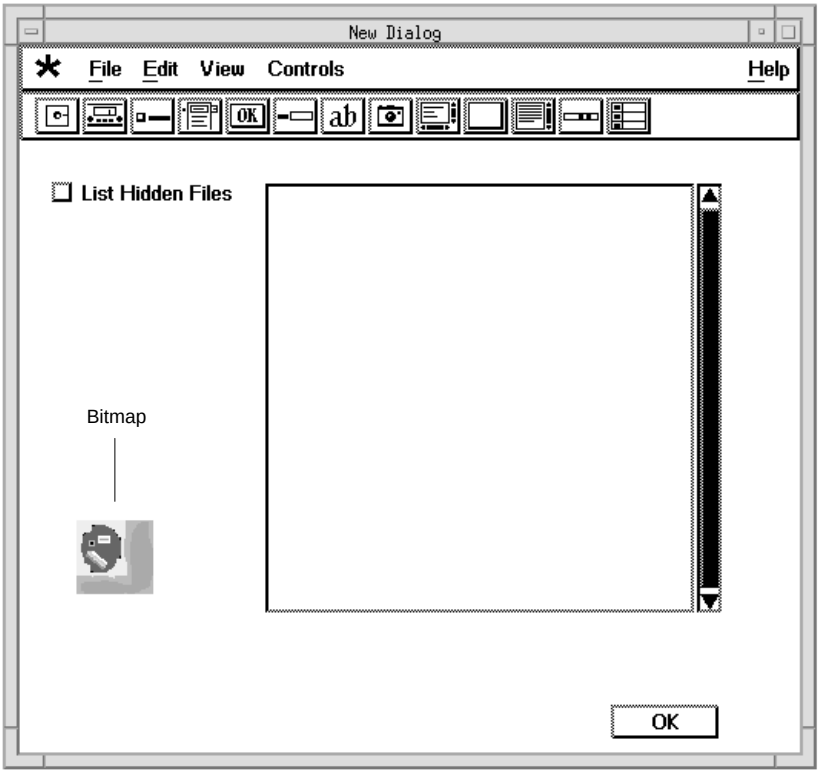
Each of these controls has attributes that can be changed by double-clicking the control in the dialog box. All ELF controls have the following attributes:

Display	Turn on Display to make the control appear in the dialog box. Turn off Display to hide the edit box.
ID	A string used to programatically address the control from an ELF macro. Many ELF macros use the Control ID field as an argument. The ID field string is returned to the ELF macro when the control triggers an exit event.
Title	The name of the control.
X	The horizontal offset of the control in pixels. This is measured from the left boundary of the dialog box.
Y	The vertical offset of the control in pixels. This is measured from the top boundary of the dialog box.

Other attributes are available for each control. These are described in the sections that follow.

Bitmaps

A bitmap is a decorative control that has no programmable features. You can create a bitmap with the Bitmap Editor, which is described in Chapter 11, "The Bitmap Editor" or with Applix Graphics.



To add a bitmap to a dialog box, follow these steps:

1. Open a dialog box.

2. Choose **Controls → Bitmap**. The Select Bitmap dialog box appears.

   The Select Bitmap dialog box displays files in the ELF search path that have an .im or .ag extension. The extension is not shown in the list box.

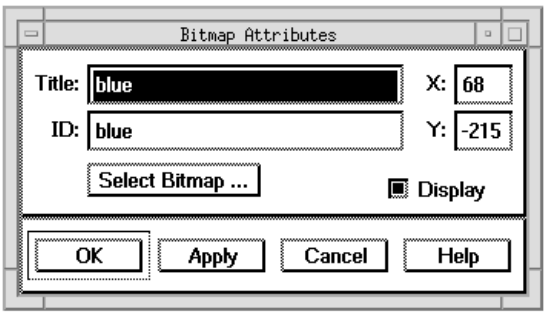
3. Highlight a bitmap file and choose **Select**.

4. Click in the dialog box.

The bitmap you selected appears in the dialog box. The following section describes how to manipulate bitmap attributes.

Bitmap Attributes

When you double-click a bitmap, the Bitmap Attributes dialog appears:

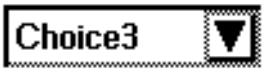


The Title, ID, Display, X, and Y attributes are common to all ELF controls. These are described in the section "Introduction to ELF Controls", earlier in this chapter. In addition, the Bitmap Attributes dialog allows you to set the following attribute:

Select Bitmap	Click Select Bitmap to select a new bitmap file from the file system.

Combo Box

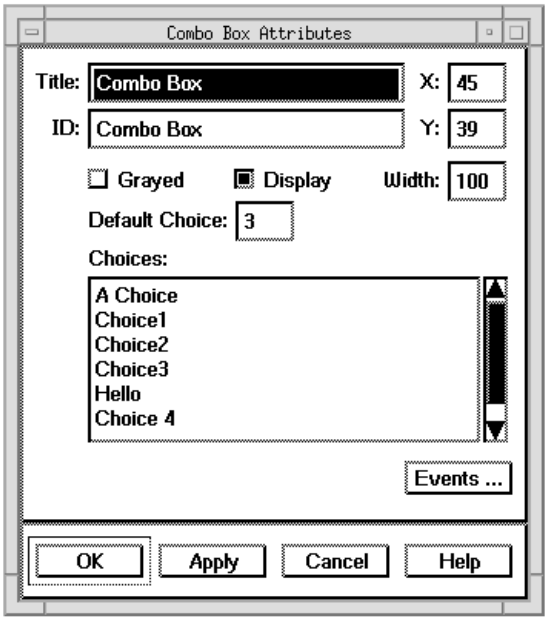
A combo box is a control that allows you to present a list of choices to a user, much like a list box. However, the combo box does not use as must of the screen as a list box, since when it is not selected, it displays only the currently-selected option.



To add a combo box to the dialog box, choose Controls → Combo Box and click in the dialog box. The combo box appears in the dialog box. The following section describes how to manipulate combo box attributes.

Combo Box Attributes

When you double-click the combo box, the combo box dialog box appears:



The Title, ID, Display, X, and Y attributes are common to all ELF controls. These are described in the section "Introduction to ELF Controls,"

earlier in this chapter. In addition, the Combo Box Attributes dialog allows you to set the following attributes:

Choices — This list of choices appears in the combo box when it is clicked by the user.

Default Choice — This is the combo box option that is visible when the application starts.

Events — Combo box events are described in the following section.

Grayed — Turn on Grayed to make the combo box grayed. Turn off Grayed to make the combo box normal.

Width — The width of the combo box is displayed in characters in the Width entry box. To change the width, enter a new value in this field.

Editable Combo Boxes

By default, a combo box is not editable. Users can click the combo box, and select entries, but they cannot directly enter strings into the control.

An editable combo box allows you to type strings directly into the combo box, the same way you would enter information in an entry field. The following macros are important when you are using an editable combo box:

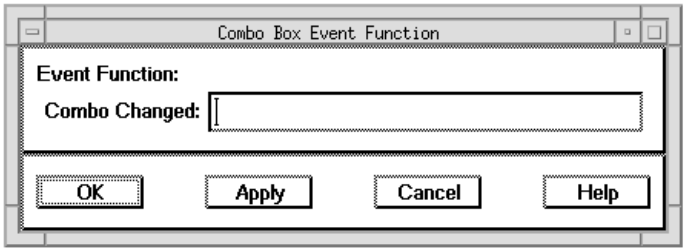
- DB_COMBO_EDITABLE@ makes a combo box editable.
- DB_COMBO_GET_EDITABLE@ returns TRUE if a combo box is editable.

- DB_CTRL_GET_VALUE@ returns the string you enter in an editable combo box. In a standard combo box, DB_CTRL_GET_VALUE@ returns the index of the selected value.

Combo Box Events

The Combo Box Attributes dialog has optional event functions for Activation. This allows you to establish a macro to run when the combo box is changed from one option to another. To establish a Combo Changed macro, follow these steps:

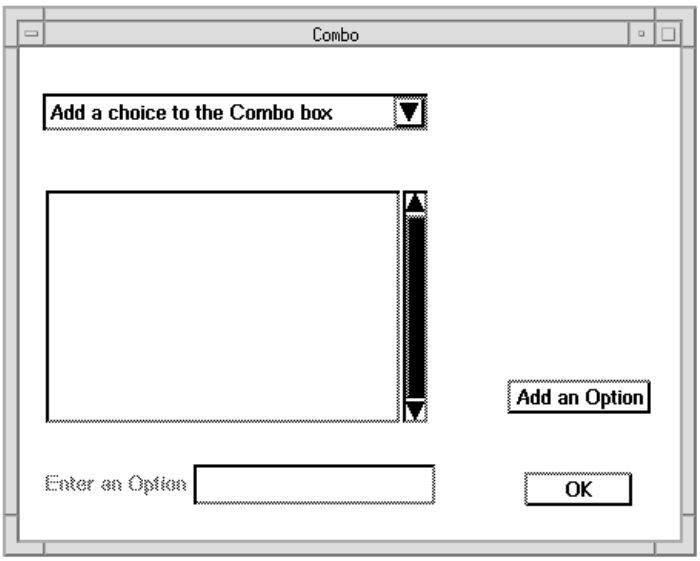
1. From the Combo Box Attributes dialog box, click Events. The Combo Box Event Function dialog box appears.



2. Enter the name of an ELF macro in the Combo Changed entry area. Two arguments are passed to this macro: the dialog box handle returned by DB_LOAD@, and the ID of the edit box which received the event.

This feature is useful if you have many combo boxes in an application, and you want a single piece of code to execute whenever one of those combo boxes is changed.

Combo Box Example




```
MACRO combo

    VAR dbox,exit_cond, ctrl_value, done, stringList,
        new_array, entry_string

    dbox = DB_LOAD@("combo.d")
    DB_CTRL_RETURN_ON_CHANGE@(dbox,"Combo Box",TRUE)
    done = FALSE

    WHILE NOT done

        DB_DISPLAY@(dbox)
        exit_cond = DB_EXIT_CTRL@(dbox)

        CASE of exit_cond

            CASE "Combo Box"
                If DB_CTRL_VALUE@(dbox, "Combo Box") = 0
                {
                        DB_CTRL_GRAYED@(dbox, "Button", FALSE)
                        DB_CTRL_GRAYED@(dbox, "Entry Box", FALSE)
```

```
                }
                If DB_CTRL_VALUE@(dbox, "Combo Box") = 1
                {
                        new_array = DB_CTRL_GET_STRINGS@(dbox, "Combo Box")
                        DB_CTRL_STRINGS@(dbox, "List Box", new_array)
                }

        CASE "Button"
                {
                        entry_string = DB_CTRL_GET_VALUE@(dbox,"Entry Box")
                        new_array = DB_CTRL_GET_STRINGS@(dbox, "Combo Box")
                        new_array = array_insert@(new_array, entry_string,
                                array_size@(new_array))
                        DB_CTRL_STRINGS@(dbox, "Combo Box", new_array)
                        DB_CTRL_GRAYED@(dbox, "Button", TRUE)
                        DB_CTRL_GRAYED@(dbox, "Entry Box", TRUE)
                }

        CASE "OK"
                done = TRUE

    ENDCASE

  WEND

ENDMACRO
```

Related Macros

The following macros are useful when programming combo boxes:

- DB_CTRL_VALUE@ sets the value of the selected option in the combo box. When a combo box is first initialized, the first item is selected.
- DB_CTRL_STRINGS@ adds strings to the combo box. For example, if string_array contains ten elements, the combo box will list ten options.
- DB_CTRL_GET_VALUE@ gets the numeric value of the selected item in the combo box. Combo box item numbers are zero-based, so the first item is 0, the second item is 1, and so on. If the combo

box is editable, DB_CTRL_GET_VALUE@ returns the string in the combo box entry area.

- DB_CTRL_GET_STRINGS@ gets an array of strings from the combo box.
- DB_COMBO_EDITABLE@ changes the mode of the combo box between editable and non-editable.
- DB_COMBO_GET_EDITABLE@ returns TRUE if the combo box is editable.

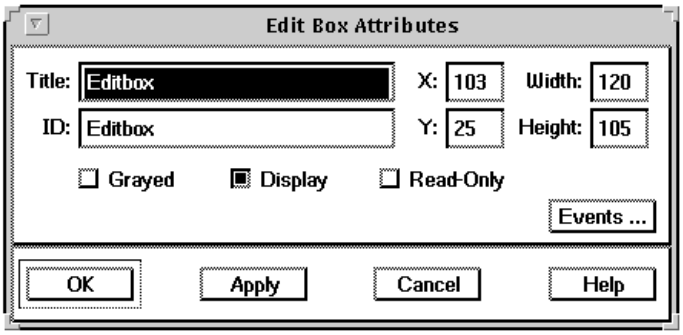
Edit Boxes

An edit box is a control that allows you to enter multiple lines of text from the keyboard, as shown in the following:

To add an edit box to the dialog box, choose Controls → Edit Box and click in the dialog box. The edit box appears in the dialog box. The following section describes how to manipulate edit box attributes.

Edit Box Attributes

When you double-click an edit box, the Edit Box Attributes dialog appears:



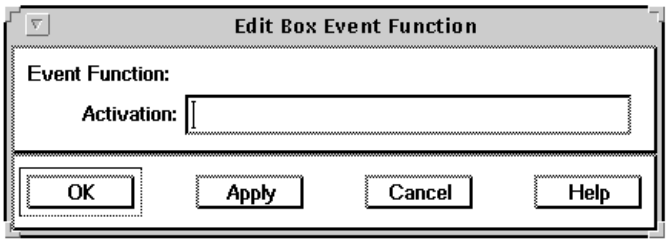
The Title, ID, Display, X, and Y attributes are common to all ELF controls. These are described in the section "Introduction to ELF Controls", earlier in this chapter. In addition, the Edit Box Attributes dialog allows you to set the following attributes:

Events	Edit box events are described in the following section.
Grayed	Turn on Grayed to make the edit box grayed. Turn off Grayed to make the edit box normal.
Height	The height of the edit box is displayed in pixels in the Height entry box. To change the height, enter a new value in this field.
Read-Only	Turn on Read-Only to make the edit box control read-only. Turn off Read-only to allow changes to the edit box control.
Width	The width of the edit box is displayed in characters in the Width entry box. To change the width, enter a new value in this field.

Edit Box Events

The Edit Box Attributes dialog has optional event functions for Activation. This allows you to establish a macro to run when the edit box gains focus. To establish an activation macro, follow these steps:

1. From the Edit Box Attributes dialog box, click **Events**. The Edit Box Event Function dialog box appears.



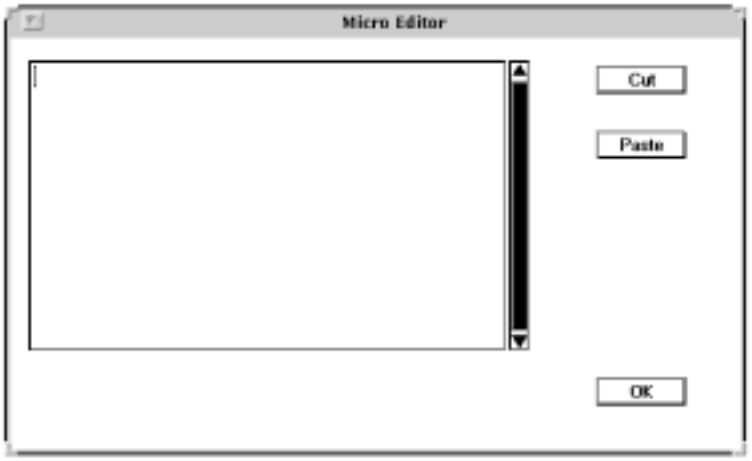
2. Enter the name of an ELF macro in the Activation entry area. Two arguments are passed to this macro: the dialog box handle returned by DB_LOAD@, and the ID of the edit box which received the event.

This feature is useful if you have many edit boxes in an application, and you want a single piece of code to execute whenever one of those edit boxes becomes active.

Edit Box Example

This example shows a dialog box containing an edit box and three buttons. You can type text into the edit box. The Cut button removes highlighted text from the box and places it in the clipboard. The Paste button takes text from the clipboard and enters it at the cursor.

The dialog box and the code for this example follow.



```
MACRO editor

VAR dbox, exit_cond, ctrl_value, done, Edit_Box_Array, Loopcounter, x, Char_Count

'load the dialog box into memory

      dbox = DB_LOAD@("editor")

'initialization section for controls before being displayed

      DB_CTRL_RETURN_ON_CHANGE@(dbox,"Cut",TRUE)
      DB_CTRL_RETURN_ON_CHANGE@(dbox,"Paste",TRUE)

      done = FALSE
      WHILE NOT done

'display dialog box to screen

      DB_DISPLAY@(dbox)

'get the id for control causing exit condition

      exit_cond = DB_EXIT_CTRL@(dbox)

'process accordingly based on exit condition

      CASE of exit_cond

            CASE "OK"
            done = TRUE

            CASE "Cut"
            DB_CTRL_CUT@(dbox)

            CASE "Paste"
            DB_CTRL_PASTE@(dbox)

      ENDCASE

ENDMACRO
```

Related Macros

The following macros are useful when programming edit boxes:

- DB_EDITBOX_SET_DATA@ sets text in the edit box.
- DB_EDITBOX_GET_DATA@ returns the contents of the edit box.
- DB_EDITBOX_CLEAR@ deletes the contents of an edit box.
- DB_EDITBOX_SELECTION@ highlights a set of characters in an edit box.
- DB_EDITBOX_GET_SELECTION@ returns the set of highlighted characters in an edit box.

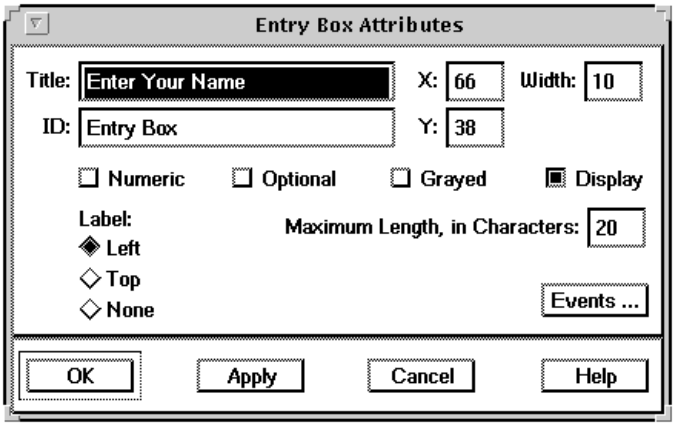
Entry Boxes

An entry box is a control that allows users to enter a single line of text from the keyboard. An entry box looks like this:

To add an entry box to the dialog box, choose Controls → Entry Box and click in the dialog box. The entry box appears in the dialog box. The following section describes how to manipulate entry box attributes.

Entry Box Attributes

When you double-click an entry box, the Entry Box Attributes dialog appears:



The Title, ID, Display, X, and Y attributes are common to all ELF controls. These are described in the section "Introduction to ELF Controls", earlier in this chapter. In addition, the Entry Box Attributes dialog allows you to set the following attributes:

Events — Entry box events are described in the following section.

Grayed — Turn on Grayed to disable the entry box and make it display in gray. Turn off Grayed to enable the entry box and make it display normally.

Label — Set the title position to the Left of the entry area, on top of the entry area, or select None to display no title at all.

Maximum Length, in Characters — Type the maximum length of input.

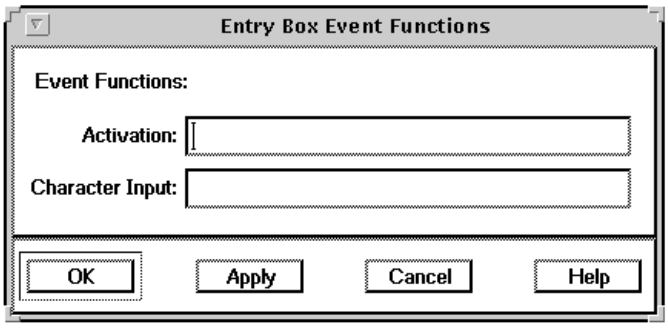
Numeric — Turn on Numeric to allow only numbers to be entered in the entry box. Turn off Numeric to allow any character to be entered.

Optional	Turn on Optional to make input in the entry box optional. Turn off Optional to make input in the entry box mandatory. If an entry area is mandatory, you must enter something in the box before you press OK. If you do not, an error is thrown.
Width	The width of the entry box entry area is displayed in characters in the Width entry box. To change the width of the entry box entry area, enter a new value.

Entry Box Events

The Entry Box Attributes dialog has optional event functions for Activation and Character Input. This allows you to establish macros that run when the entry box gains focus, or when characters are entered into the entry box. To establish macros for either of these events, follow these steps:

1. From the Entry Box attributes dialog box, click **Events**. The Entry Box Event Functions dialog box appears. The following example shows this dialog box with the names of two macros entered: Activate and Char_input.

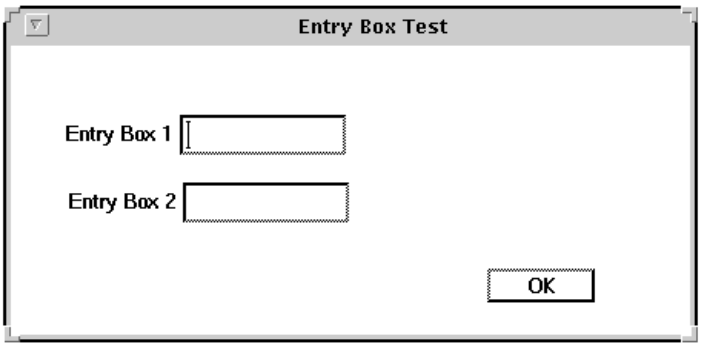


2. Enter the name of an ELF macro in either the Activation or Character Input entry area. Two arguments are passed to these macros: the dialog box handle returned by DB_LOAD@, and the ID of the Entry Box that received the event.

   The Activation macro is called when the entry box becomes active. The Character Input macro executes whenever you type a character in the entry box.

Entry Box Example

This example shows a dialog box containing two edit boxes. Both edit boxes have the Activate and Character Input events established. When either of the entry boxes is activated, the activate macro runs. When a character is typed into either of the boxes, the char_input macro runs. The dialog box and the code for this example follow.




```
macro Entry_test

/*
*      Declare Variables and Load the dialog box
*/
var dbox, exit_cond
dbox = DB_LOAD@("Entry_test.d")

/*
*      Since the Entry Box events have been enabled
```

```
*       in the dialog box, the only exit condition to test
*       for is when the OK button is pressed.
*/

while exit_cond <> "OK"

        DB_DISPLAY@(dbox)
        exit_cond = DB_EXIT_CTRL@(dbox)

WEND
endmacro

/*
*       This macro runs when one of the entry boxes
*       is activated. An entry box is activated when the
*       dialog is initially displayed, and when you move the
*       cursor from one entry box to another with the mouse.
*/
macro activate(dbox, ID)

        info_message@("Activate Event!!")

endmacro

/*
*       This macro runs when a character is entered into either
*       of the Entry boxes.
*/

macro char_input(dbox, ID)

        info_message@("Char Input Event!!")

endmacro
```

DB_CTRL_TYPING_RETURN@

DB_CTRL_TYPING_RETURN@ establishes an exit condition for an entry box. This macro works similarly to the char_input entry box event.

When you use DB_CTRL_TYPING_RETURN@, each time a character is typed in the entry box, the dialog box is suspended and control returns to the ELF macro. The following code works exactly the same as

the entry box example shown earlier in this section, but uses DB_CTRL_TYPING_RETURN@ instead of the char_input event.

```
macro Entry_test

/*
*      Declare Variables and Load the dialog box
*/
var dbox, exit_cond, val1, val2223
dbox = DB_LOAD@("Entry_test.d")

DB_CTRL_TYPING_RETURN@(dbox, "Entry Box 1", true)
DB_CTRL_TYPING_RETURN@(dbox, "Entry Box 2", true)

/*
*      This time, the Entry Box Activate event is enabled
*      in the dialog box, but the Character Input event has been
*      disabled. Character Input is handled in the While loop that follows.
*/

while exit_cond <> "OK"

      DB_DISPLAY@(dbox)
      exit_cond = DB_EXIT_CTRL@(dbox)
      if exit_cond = "Entry Box 1"
      {
      val1 = DB_CTRL_GET_VALUE@(dbox, "Entry Box 1")
      info_message@("You entered "++val1++" in Entry Box 1")
      }
      if exit_cond = "Entry Box 2"
      {
      val2 = DB_CTRL_GET_VALUE@(dbox, "Entry Box 2")
      info_message@("You entered "++val2++" in Entry Box 2")
      }


WEND
endmacro

/*
*      This macro runs when one of the entry boxes
*      is activated. An entry box is activated when the
*      dialog is initially displayed, and when you move the
*      cursor from one entry box to another with the mouse.
*/
macro activate
      info_message@("Activate Event!!")
endmacro
```

Related Macros

The following macros are useful for programming entry boxes:

- DB_CTRL_VALUE@ sets the string in the entry box.
- DB_CTRL_LENGTH@ sets the maximum number of characters that can be typed in the entry box.
- DB_CURSOR_IN_ENTRY@ sets the cursor in the entry box, and highlights characters in the entry box.
- DB_CTRL_GET_VALUE@ returns the string entered in the entry box.
- DB_CTRL_VALID_CHARS@ limits the type of characters that can be typed in an entry box. If you attempt to type a character other than those allowed for the entry box, the system beeps and the character is not placed in the entry box. The format for DB_CTRL_VALID_CHARS@ is:

  DB_CTRL_VALID_CHARS@(dbox, control_ID, valid_chars)

  valid_chars is a string containing all the characters that can validly be entered in the entry box specified by control_ID.

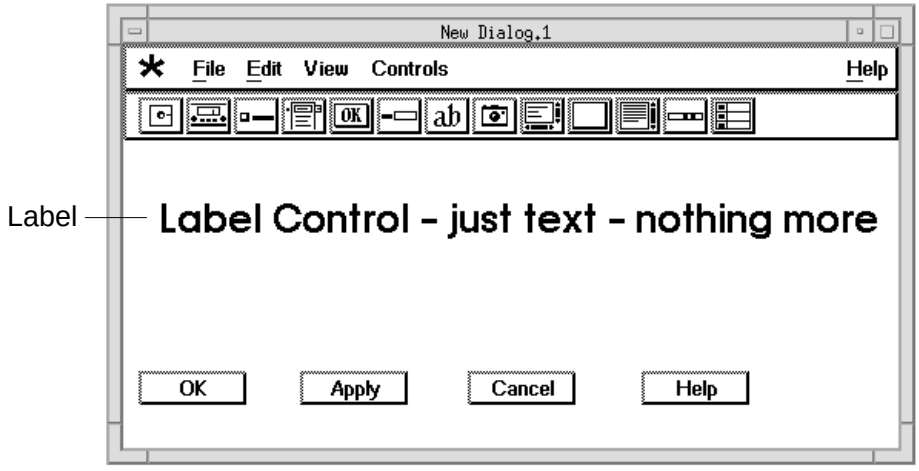
  For example, the following macro allows only whole numbers to be typed in the entry box:

  DB_CTRL_VALID_CHARS@(dbox,"Age","0123456789")

  DB_CTRL_VALID_CHARS@ does not offer the user any information as to what is a valid character. This macro should be used cautiously, and the user should be notified as to what characters are valid, and which are not valid.

Labels

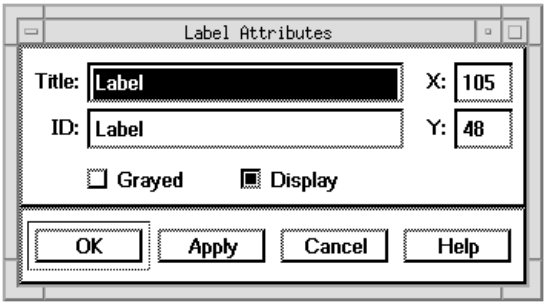
A label is a control that has no programmable features. Its purpose is to display a text string in a dialog box. To add a label to a dialog box, choose Controls → Label, and click in the dialog box. The label appears in the dialog box.



The following section describes how to manipulate label attributes.

Label Attributes

When you double-click a label, the Label Attributes dialog appears:



The Title, ID, Display, X, and Y attributes are common to all ELF controls. These are described in the section "Introduction to ELF Controls", earlier in this chapter. In addition, the Label Attributes dialog allows you to set the following attribute:

Grayed	Turn on Grayed to make the label grayed. Turn off Grayed to make the label display normally.

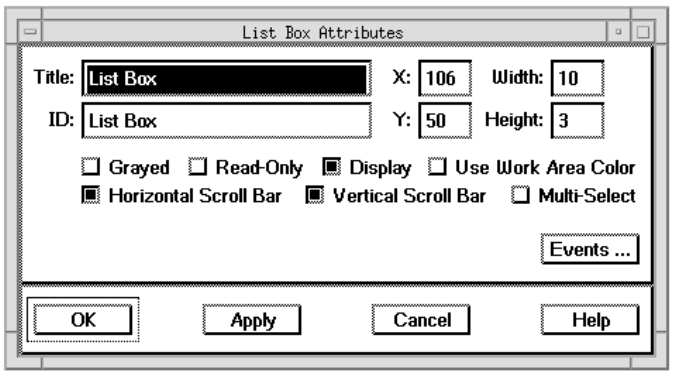
List Boxes

A list box is a control that contains a list, and allows you to select one or more entries from the list. A list box looks like this:

To add a list box to the dialog box, choose Controls → List Box and click in the dialog box. The list box appears in the dialog box. The following section describes how to manipulate list box attributes.

List Box Attributes

When you double-click a list box, the List Box Attributes dialog appears:



The Title, ID, Display, X, and Y attributes are common to all ELF controls. These are described in the section "Introduction to ELF Controls," earlier in this chapter. In addition, the List Box Attributes dialog box allows you to set the following attributes:

Events — List box events are described in the following section.

Grayed — Turn on Grayed to disable the list box, and make it display in gray. Turn off Grayed to enable the list box and make it display normally.

Horizontal Scroll bar — Turn on Horizontal Scroll Bar to place a horizontal scroll bar on the list box. Turn off Horizontal Scroll Bar to remove the horizontal scroll bar from the list box.

Multi-Select — Turn on Multi-Select to allow the user to select more than one entry from the list box. Turn off

Multi-Select to allow only one entry to be selected at a time.

Read-Only — Turn on Read-Only to make the list box read-only. Turn off Read-Only to allow changes to the list box text.

Use Work Area Color — Turn on Use Work Area Color to use work area colors in the list box. Work Area Colors can be changed through the Color Preferences dialog box. Turn off Force All Text Entry Widgets to Use Work Area Background Color to use the regular dialog box background color in the list box.

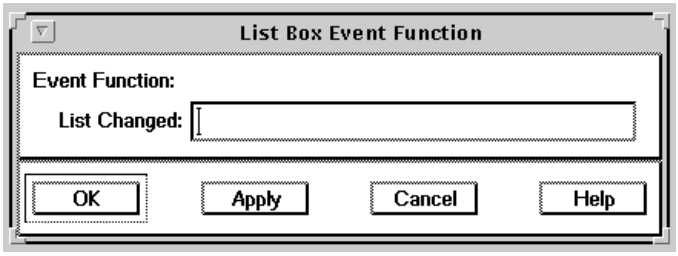
Vertical Scroll Bar — Turn on Vertical Scroll Bar to place a vertical scroll bar on the list box. Turn off Vertical Scroll Bar to remove the vertical scroll bar from the list box.

Width/Height — The width and height of the list box are displayed (in characters) in the Width and Height entry boxes. To change the width and height of the list box, type new dimensions in the appropriate entry boxes.

List Changed Event

The List Box Attributes dialog has an optional List Changed Event Function option. This allows you to establish a macro that runs whenever you click an item in the list box. To establish a list changed event macro, follow these steps:

1. From the List Box Attributes dialog box, click Events. The List Box Event Function dialog box appears.



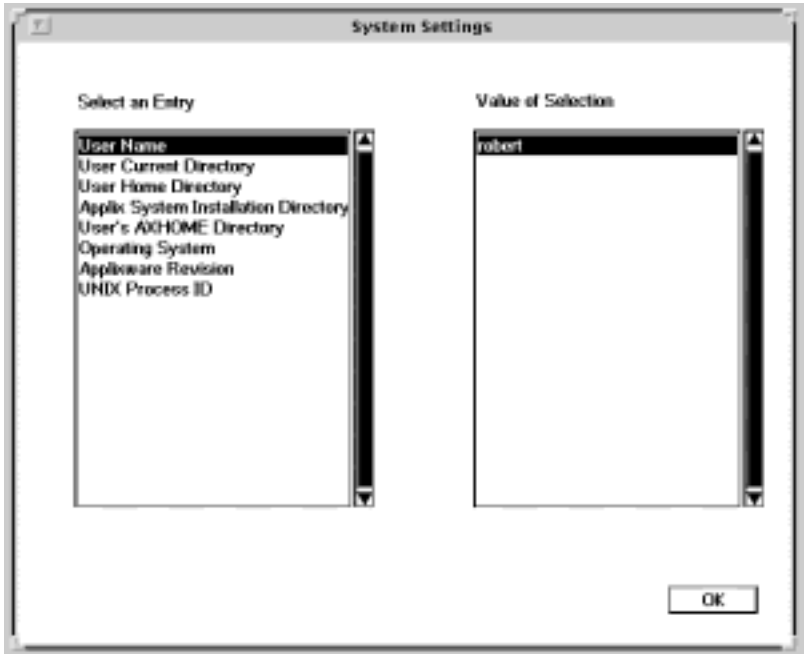
2. Enter the name of an ELF macro in the List Changed entry box. This macro executes whenever you click an item in the list box. Two arguments are passed to this macro: the dialog box handle returned by DB_LOAD@, and the ID of the list box that received the event.

This feature is useful if you have many list boxes in an application, and you want a single piece of code to execute whenever one of those list boxes is clicked. The processing of this macro supercedes the execution of the exit condition loop, as shown in the following example.

Macro Code	Event Code
MACRO ListBoxEventTest /* * 1. Declare Variables * 2. Load the dialog box * 3. Establish exit conditions. * * The loop below is used to * process exit conditions. Because the * list box has the List Changed * event defined, the tests for the List * Box exit condition are never executed. */ while exit_cond <> "OK" DB_DISPLAY@(dbox) exit_cond = DB_EXIT_CTRL@(dbox) if exit_cond = "List Box" { **/*** *** This code is superceded by the** *** ListBoxEvent Macro.** ***/** } WEND ENDMACRO	**/*** *** This code is executed when the List Box** *** is clicked.** ***/** MACRO ListBoxEvent(dbox, ListID) info_message@("You clicked the List Box!") ENDMACRO

List Box Example

The following example shows two list boxes. The list box on the left contains eight entries with the names of various system parameters. The list box on the right displays the system parameter selected. The dialog box and code follow.




```
macro System_Settings

/*
* Declare variables
*/
var exit_cond, dbox, Parameter_array, UserName
var CurrentDirectory, UserHomeDirectory, ApplixInstallDirectory, AxhomeDirectory, OS
var UnixProcessId, ListBoxValue, ApplixVersion

/*
* Load the dialog box and establish exit conditions.
*/

dbox = DB_LOAD@("System_Settings.d")
DB_CTRL_RETURN_ON_CHANGE@(dbox, "List Box", TRUE)


Parameter_array = "User Name", "User Current Directory", "User Home Directory", "Applix System
Installation Directory", "User's AXHOME Directory", "Operating System", "Applixware Revision",
"UNIX Process ID"
```

```
/*
* Only arrays can be loaded into a List Box using the DB_CTRL_STRINGS@
* function. Therefore, add an empty string to the user name, the current directory,
* and so on.
*/
UserName = User_Name@(), " "
CurrentDirectory = Current_Dir@(), " "
UserHomeDirectory = User_Dir@(), " "
ApplixInstallDirectory = System_Dir@()," "
AxhomeDirectory = Axhome_Dir@(), " "
OS = SHELL_COMMAND@("uname")
ApplixVersion = VERSION_String@(), " "
UnixProcessId = UNIX_PROCESS_ID@(), " "

  DB_CTRL_STRINGS@(dbox, "List Box", Parameter_array)
  DB_CTRL_STRINGS@(dbox, "List Box.1", UserName)


while exit_cond <> "OK"

  DB_DISPLAY@(dbox)
  exit_cond = DB_EXIT_CTRL@(dbox)
  ListBoxValue = DB_CTRL_GET_VALUE@(dbox, "List Box")

  if exit_cond = "List Box" then
  {
    if ListBoxValue = 0 then DB_CTRL_STRINGS@(dbox, "List Box.1", UserName)
    if ListBoxValue = 1 then DB_CTRL_STRINGS@(dbox, "List Box.1",
         CurrentDirectory)
    if ListBoxValue = 2 then DB_CTRL_STRINGS@(dbox, "List Box.1",
         UserHomeDirectory)
    if ListBoxValue = 3 then DB_CTRL_STRINGS@(dbox, "List Box.1",
         ApplixInstallDirectory)
    if ListBoxValue = 4 then DB_CTRL_STRINGS@(dbox, "List Box.1",
         AxhomeDirectory)
    if ListBoxValue = 5 then DB_CTRL_STRINGS@(dbox, "List Box.1", OS)
    if ListBoxValue = 6 then DB_CTRL_STRINGS@(dbox, "List Box.1",
         ApplixVersion)
    if ListBoxValue = 7 then DB_CTRL_STRINGS@(dbox, "List Box.1",
         UnixProcessId)
  }

WEND

endmacro
```

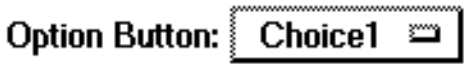
Related Macros

The following macros are useful when programming list boxes:

- DB_CTRL_VALUE@ sets the value of the selected option in the list box. When a list box is first initialized, the first item is selected. If you do not want any items selected, use DB_CTRL_VALUE@ with value set to -1. To select more than one item in a multi-select list box, use DB_CTRL_VALUE@ to pass an array of the values to be selected.
- DB_CTRL_STRINGS@ adds strings to the list box. For example, if string_array contains ten elements, the list box will list ten options.
- DB_CTRL_GET_VALUE@ gets the numeric value of the selected item in the list box. List box item numbers are zero-based, so the first item is 0, the second item is 1, and so on. If the list box is set to multi-select, DB_CTRL_VALUE@ returns an array of selected values.
- DB_CTRL_GET_STRINGS@ gets an array of strings from the list box.
- DB_CTRL_PICKABLE@ indicates whether text that appears in a list box can be selected. DB_CTRL_PICKABLE@ is useful when you want to use a list box to display informational text rather than a list of selectable items.
- DB_CTRL_MONOSPACE@ specifies that list box items are displayed using a monospace typeface. Usually, list box items are displayed using a proportional typeface. Using a monospace typeface is helpful when you are including columns in your list box text. With the monospace typeface, all character have equal width so you can align columns using the space bar.

Option Buttons

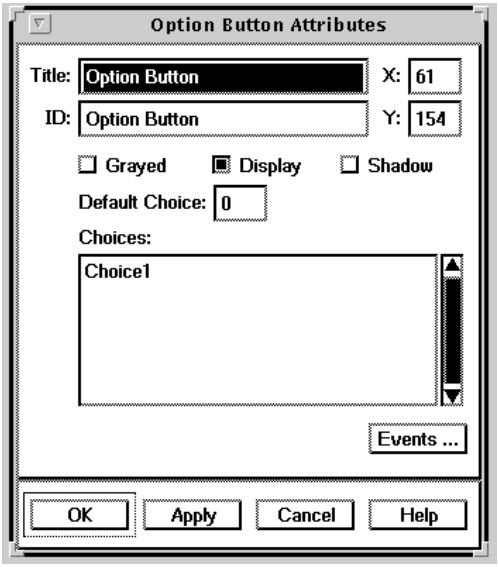
An option button is a label that is associated with a list of options:



To add an option button to the dialog box, choose Controls → Option Button and click in the dialog box. The option button appears in the dialog box. The following section describes how to manipulate option button attributes.

Option Button Attributes

If you double-click an option button, the Option Button Attributes dialog appears:



The Title, ID, Display, X, and Y attributes are common to all ELF controls. These are described in the section "Introduction to ELF Controls," earlier in this chapter. In addition, the Option Button Attributes dialog allows you to set the following attributes:

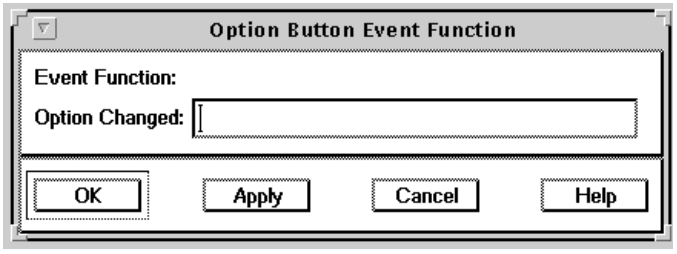
Events	Option button events are discussed in the next section.
Choices	The option button choices are displayed in the Choices list box. You can edit this list box to add as many choices as you like. Choices can also be added and deleted from the option button at run time. You can change the name of a choice by selecting a name, then typing a new name.
Default Choice	Type the number of the default option choice in the Default Choice entry box. The choices are zero-based, so the first option is number 0, the second option is 1, ands so on.

Grayed	Turn on Grayed to disable the option button and make it appear gray. Turn off Grayed to enable the option button and make it display normally.
Shadow	Turn on Shadow to make the option button display with a shadow attribute. Turn off Shadow to make the option button display normally.

Option Changed Event

You can configure a macro to run whenever the value of the option button changes. Follow these steps:

1. From the Option Button Attributes dialog box, click **Events**. The Option Button Event Function dialog appears:



2. Enter the name of an ELF macro in the Option Changed entry box. This macro executes whenever any change in the option button state occurs. Two arguments are passed to this macro: the dialog box handle returned by DB_LOAD@, and the ID of the option button that received the event.

This feature is useful if you have many option buttons in an application, and you want all of them to run a particular event handler whenever they are changed.

If you establish an option changed event macro, the processing of this macro supercedes the processing of your exit condition loop. The following code illustrates this.

Macro Code

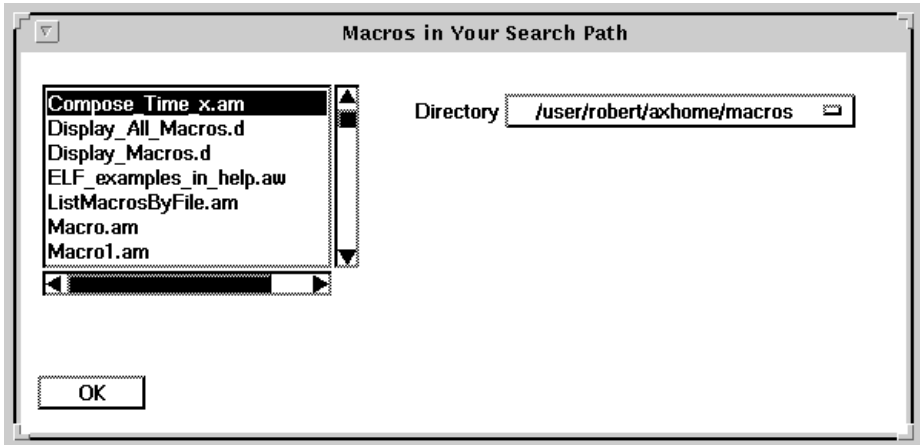
```
MACRO OptionEventTest

/*
*  1. Declare Variables
*  2. Load the dialog box
*  3. Establish exit conditions.
*
* The While loop below is used to
* process exit conditions. Because the
* Option button has the OptionChanged
* event defined, the tests for the Option
* Button exit condition are never executed.
*/

while exit_cond <> "OK"

 DB_DISPLAY@(dbox)
 exit_cond = DB_EXIT_CTRL@(dbox)

 if exit_cond = "Option Button"
  {
/*
* This code is superceded by the
* OptionEvent Macro.
*/

  }
WEND

ENDMACRO
```

Event Code

```
/*
*  This code is executed when the option
*  button is changed.
*/

MACRO OptionEvent(dbox, option_id)

info_message@("You changed the Option button!")

ENDMACRO
```

Option Button Example

The following code example shows an option button and a list box. The option button is used to display directories that are included in your elf_search_path@ system variable. The dialog box and code follow.




```
Macro X_OptionButton
/*
*     Declare Variables
*/
var dbox, exit_cond, path, elf_search_path, elements, x, files

/*
*     Load the dialog box and establish exit conditions.
*/
      dbox = DB_LOAD@("option")
      DB_CTRL_RETURN_ON_CHANGE@(dbox,"Option Button",TRUE)
/*
*     Get the ELF Search Path of the user, and load the directories
*     in the path into the Option Button.
*/
      path = SYSTEM_VAR@("elf_search_list@")
      DB_CTRL_STRINGS@(dbox, "Option Button", path)
      files = SHELL_COMMAND@("ls "++path[0])
      DB_CTRL_STRINGS@(dbox, "List Box", files)
/*
```

```
*       The While loop below is used to process exit conditions
*       from the dialog box.  Only two exit conditions are defined:
*       1. The user presses OK.
*       2. The user changes the displayed file list by using
*          the option button to display a new path.
*/
while exit_cond <> "OK"

        DB_DISPLAY@(dbox)
        exit_cond = DB_EXIT_CTRL@(dbox)

/*
*       If the user changes the value of the option button, this conditional
*       statement executes.
*/
        if exit_cond = "Option Button"
                files = SHELL_COMMAND@("ls "++
                        path[DB_CTRL_GET_VALUE@(dbox, "Option Button")])
                DB_CTRL_STRINGS@(dbox, "List Box", files)
        WEND

endmacro
```

Related Macros

The following macros are useful when programming option buttons:

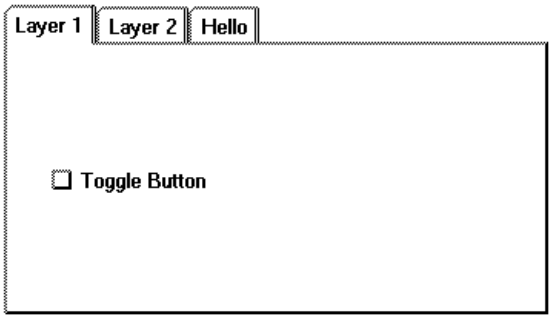
- DB_CTRL_VALUE@ sets the value of the selected option in the option button. When an option button is first initialized, the first item is highlighted. If you do not want any items highlighted, use DB_CTRL_VALUE@ with the value argument set to -1.
- DB_CTRL_STRINGS@ adds strings to the option button. For example, if string_array contains ten elements, the option button will contain ten options.
- DB_CTRL_GET_VALUE@ gets the numeric value of the selected item in the option button. Option button item numbers are zero-based, so the first item is 0, the second item is 1, and so on.

- DB_CTRL_GET_STRINGS@ gets an array of strings from the option button.

Panels

A panel is a control that has no programmable features. Its purpose is to visually separate controls on the screen to make the dialog box more presentable to the user.

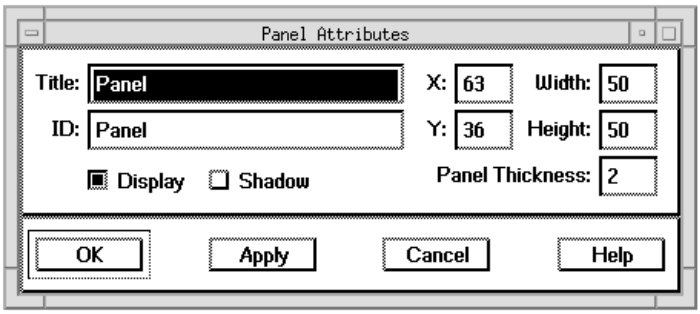
The most elaborate type of panel is the Tab control. The Tab control is a panel with more than one layer. It allows you to separate groups of controls and display them by clicking on a titled tab at the top of the control. The following shows an example of a tab control with three layers entitled Layer 1, Layer 2, and Hello:



To add a panel to a dialog box, choose Controls → Panel, and click in the dialog box. The panel appears in the dialog box. The following section describes how to manipulate panel attributes.

Panel Attributes

When you double-click a panel, the Panel Attributes dialog appears:



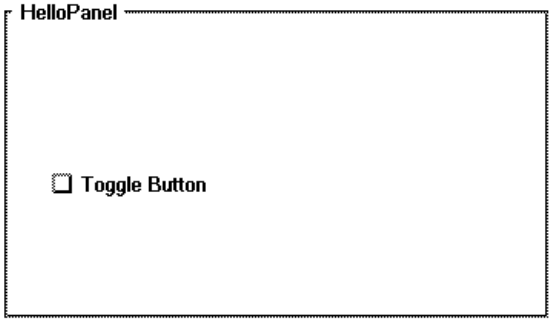
The Title, ID, Display, X, and Y attributes are common to all ELF controls. These are described in the section "Introduction to ELF Controls", earlier in this chapter. In addition, the Panel Attributes dialog allows you to set the following attributes:

Shadow — Turn on Shadow to make the panel display with a shadow attribute. Turn off Shadow to make the panel display normally.

Panel Thickness — Panel Thickness changes the look of the panel. The following shows two panels. The panel on the left has a thickness of 2. The panel on the right has a thickness of 8.

Panel Style — The Panel Style option gives you one of three options: Plain and Labeled are simple panels.

Labeled panels have a title, as shown in the following illustration:



The Layered option turns the panel into a Tab control. The Tab control is described in the following section.

Layer Names

The Layer Names list box is enabled when you select the layered option. You can type as many entries in the list box as you want. For each entry, a layer is created in the Tab control.

Tab Control

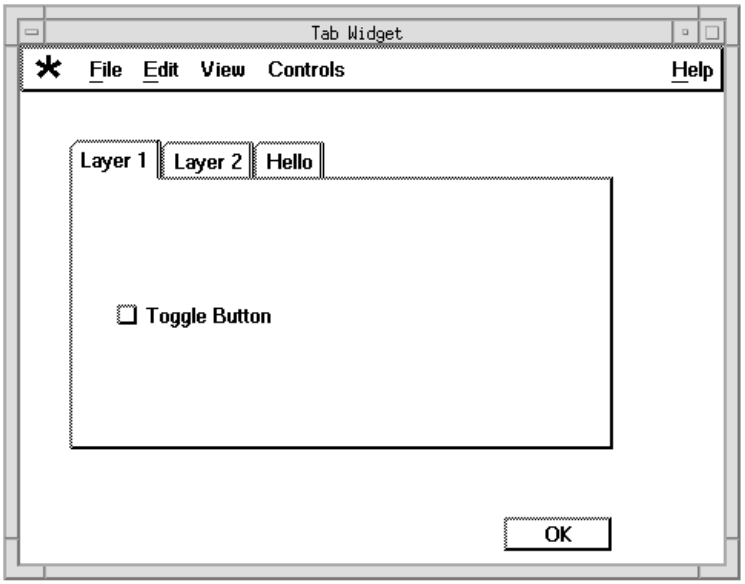
The Tab control is often used in Windows 95 applications to separate controls, and present a cleaner user interface to the user. You can add controls to any layer of the Tab control that you want. Only the controls on the top layer (also called the *active* layer), are available to the user.

The Tab control is supported in versions 4.3 or later of Applixware. If your site runs older versions of Applixware, you may want to include a revision check in any program that uses the Tab control. ELF programs that use the tab control may yield unpredictable results if run

on versions of Applixware earlier than 4.3. The example that follows includes a revision check.

Tab Control Example

The following example shows a Tab control with three layers. The program uses DB_CTRL_RETURN_ON_CHANGE@ to establish events when the user clicks from one layer to another.




```
MACRO Tab_Widget

    VAR dbox, exit_cond, ctrl_value, done

' Check the version of the software before running this macro! The software should
' be version 4.3 or later of Applixware.

    versionstring = VERSION_STRING@()
```

```
    if SUBSTRING@(versionstring,1,3) < "4.3"
    {
        info_message@("Sorry - can't run on this version!")
        return
    }

    dbox = DB_LOAD@("Tab_Widget.d")
    DB_CTRL_RETURN_ON_CHANGE@(dbox,"HelloPanel",TRUE)
    done = FALSE

    WHILE NOT done

        DB_DISPLAY@(dbox)

        IF DB_CANCELLED@(dbox)
            RETURN

        exit_cond = DB_EXIT_CTRL@(dbox)

        CASE of exit_cond

    ' The following CASE statement prints the nakme of the layer that is made
    ' active when the user clicks the Tab control.

            CASE "HelloPanel"
            INFO_MESSAGE@(DB_TABCTRL_GET_ACTIVE_LAYER@
                                        ( dbox, "HelloPanel"))

            CASE "OK"
            done = TRUE

        ENDCASE
    WEND
ENDMACRO
```

Related Macros

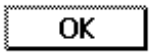
The following macros are useful when programming Tab controls:

- DB_TABCTRL_ACTIVE_LAYER@ changes the active layer of the Tab control.
- DB_TABCTRL_APPEND_LAYER@ adds a layer to the tab control.

- DB_TABCTRL_GET_ACTIVE_LAYER@ returns the name of the active layer of the Tab control.
- DB_TABCTRL_INSERT_CONTROL@ associates a control to a layer of a tab control.

Push Buttons

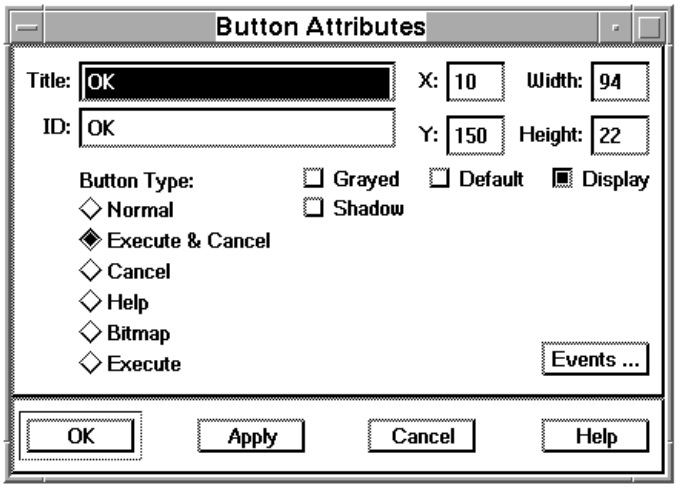
Push buttons allow a user to start or cancel processing in a dialog box. A push button looks like this:



To add a push button to the dialog box, choose Controls → Push Button and click in the dialog box. The push button appears in the dialog box. The following section describes how to manipulate push button attributes.

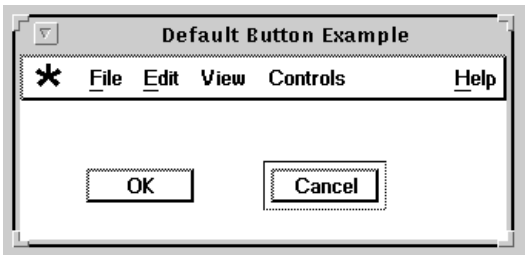
Push Button Attributes

If you double-click a push button, the Button Attributes dialog appears:



The Title, ID, Display, X, and Y attributes are common to all ELF controls. These are described in the section "Introduction to ELF Controls", earlier in this chapter. In addition, the Button Attributes dialog allows you to set the following attributes:

Default — Click Default to make the push button the default for the dialog box. You can have only one default push button in a dialog box. When you select a push button as the default, it displays in the dialog box surrounded by a rectangle. In the following example, the Cancel button is the default push button:



Events	Push button events are discussed in the next section.
Grayed	Turn on Grayed to disable the push button. The button displays on the screen in gray to indicate that it cannot be pressed by the user. Turn off Grayed to enable the push button.
Shadow	Turn on Shadow to make the push button display with a shadow attribute. Turn off Shadow to make the option button display normally.

Button Types

The Button Types attribute allows you to configure a push button as one of the following types:

Cancel	Cancel buttons can be chosen by pressing ESC or CANCEL, or by clicking on the button with the mouse. A Cancel button dismisses the dialog box.
Execute	Execute buttons are the same as Normal buttons. They generate an exit condition. They do not dismiss the dialog box. They are not associated with any keys.
Execute & Cancel	A push button of type Execute & Cancel accepts dialog box entries and exits the dialog box when it is pressed. Note that while buttons of type Execute & Cancel often contain the text "OK," you can use any text in the button. For example, you might want "Delete" to be a push button of type Execute & Cancel.

Help	A push button of type Help displays a Help window when it is clicked.
Normal	Normal buttons are the same as Execute buttons. They generate an exit condition. They do not dismiss the dialog box. They are not associated with any keys.
Bitmap	A push button of type Bitmap is a button that does not contain text. Usually, when you define a bitmap button, you place a bitmap picture on the button. Therefore, the action associated with the push button can be associated with the bitmap picture.

The Default Push Button

One push button in a dialog box is designated the *default* push button. The default push button is the one that is automatically selected when RETURN is pressed.

When you create a dialog box using the dialog box editor, the OK button that is automatically included in the dialog box is the default button. If you change the default button, the default designation is automatically removed from the previous default button.

You can change the default push button through the dialog box editor, or through an ELF macro. To make a button the default using the dialog box editor, follow these steps:

1. Double-click on the button.
2. Turn on the **Default** option in the Button Attributes dialog box.
3. Click **OK**.

DB_CTRL_DEFAULT_BUTTON@ changes the default button in a dialog box, or removes the default designation from a push button if you do not want any buttons in the dialog box to be a default push button.

The format for the macro is:

```
DB_CTRL_DEFAULT_BUTTON@(dbox, control_ID, value)
```

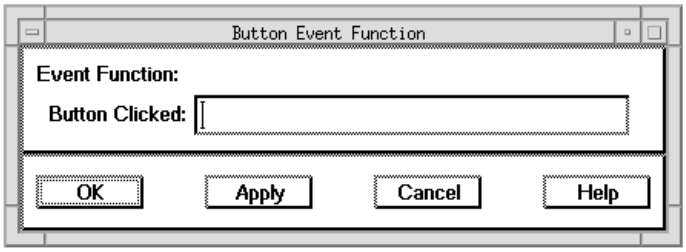
value can be TRUE or FALSE. A value of TRUE means the push button is made the default button. A value of FALSE means the push button is not the default button.

To make a push button the default button when the dialog box is first displayed, place the DB_CTRL_DEFAULT_BUTTON@ macro *before* the DB_DISPLAY@ macro in your dialog box macro.

Button Clicked Event Function

You can establish a macro to run whenever a push button is clicked. Follow these steps:

1. From the dialog box editor, double-click the push button. The Push Button Attributes dialog appears.
2. Click **Events**. The Push Button Event Function dialog appears:



3. Enter the name of an ELF macro in the Button Clicked entry box. The macro you specify executes whenever the push button is clicked. Two arguments are passed to this macro: the dialog box

handle returned by DB_LOAD@, and the ID of the push button that received the event.

This feature is useful if you have many push buttons in an application, and you want all of them to run a particular event handler whenever they are changed.

If you establish a Button Clicked event function, the execution of this macro supercedes the execution of exit condition event loop. The following example code illustrates this.

Macro Code

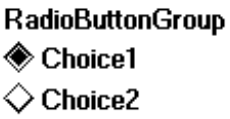
```
MACRO ButtonClickedEventTest

/*
*  1. Declare Variables
*  2. Load the dialog box
*  3. Establish exit conditions.
*/

/*
* The While loop below is used to
* process exit conditions. Because the
* push button has the ButtonClicked
* event defined, the tests for the Push
* Button exit condition are never executed.
*/

while exit_cond <> "OK"

 DB_DISPLAY@(dbox)
 exit_cond = DB_EXIT_CTRL@(dbox)

 if exit_cond = "Push Button"
  {
/*
* This code is superceded by the
* Button Clicked Macro.
*/

  }
WEND

ENDMACRO
```

Event Code

```
/*
*  This code is executed when the radio
*  button is changed.
*/

MACRO ButtonClickedEvent(dbox, ButtonID)

info_message@("You clicked the push button!")

ENDMACRO
```

Radio Button Groups

A radio button group is a control that allows a user to pick one of a set of selections. A radio button group in an ELF dialog box looks like this:



To add a radio button group to a dialog box, choose Controls → Radio Button Group and click in the dialog box. The radio button group appears in the dialog box. The following section describes how to manipulate radio button group attributes.

Radio Button Group Attributes

When you double-click a radio button group in a dialog box, the Radio Button Attributes dialog appears:

The Title, ID, Display, X, and Y attributes are common to all ELF controls. These are described in the section "Introduction to ELF Controls", earlier in this chapter. In addition, the Radio Button Attributes dialog allows you to set the following attributes:

Buttons

The radio button names are displayed in the Buttons list box. You can change the name of a button by selecting a name, then typing a new name.

To add a button to the radio button group:

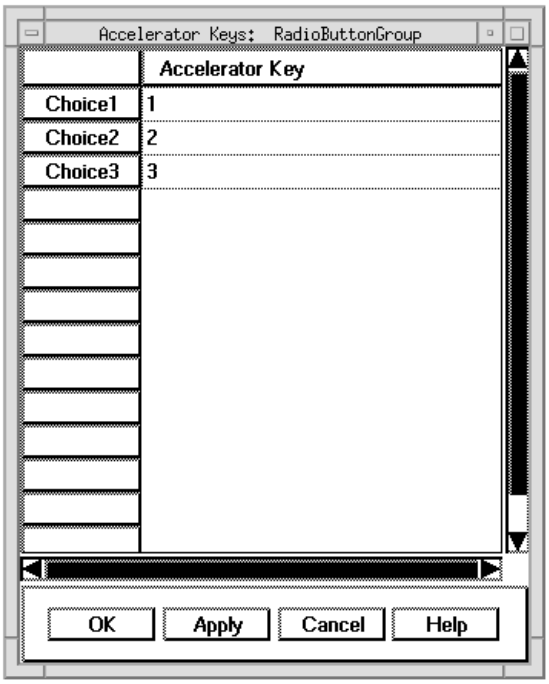
1. Place the cursor at the end of a radio button name.
2. Press **RETURN**.
3. Type the new radio button name.

	The new radio button is added when you click OK.
Events	Radio button group events are discussed in the next section.
Default Choice	Type the number of the default button choice in the Default Choice entry box. The choices are zero-based, so the first selection is number 0, the second selection is 1, and so on.
Grayed	Turn on Grayed to make the radio button group grayed. Turn off Grayed to make the radio button group normal.

Accelerator Keys

You can assign accelerator keys to elements items in a Radio Button Group. To do this, folloa these steps:

1. Double-click the radio button group.
2. Click **Accelerator Keys**. The following screen appears:



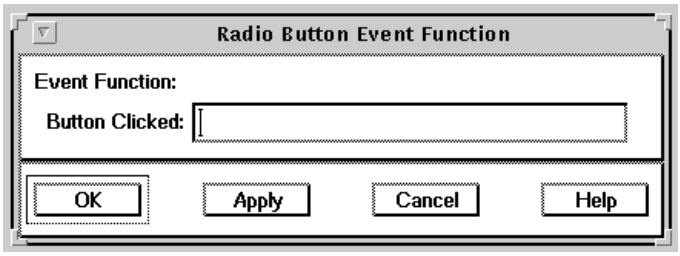
3. Click next to the choices in the list, and press the letter or number that you want to use as an accelerator. For example, in this screen, the number 1 is entered next to Choice1. This means that Choice1 will be selected when the user presses <Alt>-<1>.

4. Click OK.

Radio Button Event Function

You can establish a macro to run whenever the value of the RadioButtonGroup is changed. Follow these steps:

1. From the Radio Button Attributes dialog box click **Events**. The Radio Button Event Function dialog box appears:



2. Enter the name of an ELF macro in the Button Clicked entry box. The macro you specify executes whenever any change in the radio button state occurs. Two arguments are passed to this macro: the dialog box handle returned by DB_LOAD@, and the ID of the radio button that received the event.

This feature is useful if you have many radio button groups in an application, and you want all of them to run a particular event handler whenever they are changed.

If you establish a button clicked event function, the execution of this macro supercedes the execution of exit condition event loop. The following example code illustrates this.

Macro Code

```
MACRO RadioButtonEventTest

/*
*  1. Declare Variables
*  2. Load the dialog box
*  3. Establish exit conditions.
*/

/*
* The While loop below is used to
* process exit conditions. Because the
* radio button has the ButtonClicked
* event defined, the tests for the Radio
* Button exit condition are never executed.
*/

while exit_cond <> "OK"

 DB_DISPLAY@(dbox)
 exit_cond = DB_EXIT_CTRL@(dbox)

 if exit_cond = "Radio Button"
  {
/*
* This code is superceded by the
* Button Clicked Macro.
*/

  }
WEND

ENDMACRO
```

Event Code

```
/*
*  This code is executed when the radio
*  button is changed.
*/

MACRO ButtonClickedEvent(dbox, RadioID)

info_message@("You clicked the radio button!")

ENDMACRO
```

Related Macros

The following macros are useful when programming radio buttons:

- DB_CTRL_VALUE@ sets the value of the selected option in the radio button. When a radio button is first initialized, the first item is highlighted. If you do not want any items highlighted, use DB_CTRL_VALUE@ with the value argument set to -1.
- DB_CTRL_STRINGS@ adds strings to the radio button. For example, if string_array contains ten elements, the radio button will contain ten options.
- DB_CTRL_GET_VALUE@ gets the numeric value of the selected item in the radio button. Radio button item numbers are zero--based, so the first item is 0, the second item is 1, and so on.
- DB_CTRL_GET_STRINGS@ gets an array of strings from the radio button.

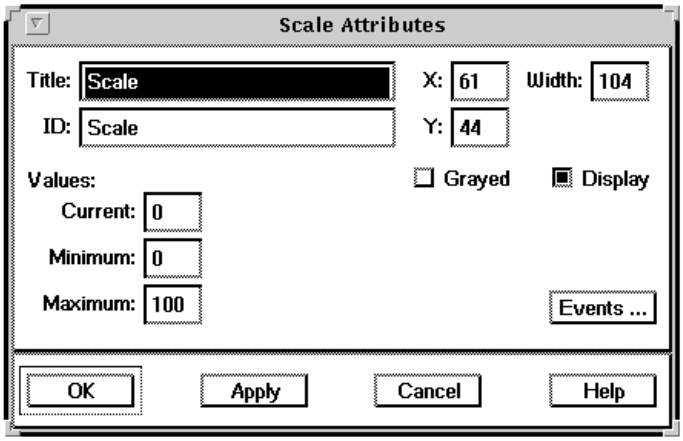
Scale

A scale is a control that allows you to select a numerical value by moving a slider between a maximum and a minimum value. A scale looks like this:

To add an entry box to the dialog box, choose Controls → Scale and click in the dialog box. The scale appears in the dialog box. The following section describes how to manipulate scale attributes.

Scale Attributes

When you double-click a scale, the Scale Attributes dialog appears:



The Title, ID, Display, X, and Y attributes are common to all ELF controls. These are described in the section "Introduction to ELF Controls", earlier in this chapter. In addition, the Scale Attributes dialog allows you to set the following attributes:

Current	The current position of the scale slider in the scale area appears in the Current entry box. To change the current position of the scale slider, type the value in the entry box.
Events	Scale events are discussed in the next section.
Grayed	Turn on Grayed to make the edit box grayed. Turn off Grayed to make the edit box normal.
Maximum	The maximum value of the scale appears in the Maximum entry box. This is the value of the scale if you move the slider all the way to the right. To

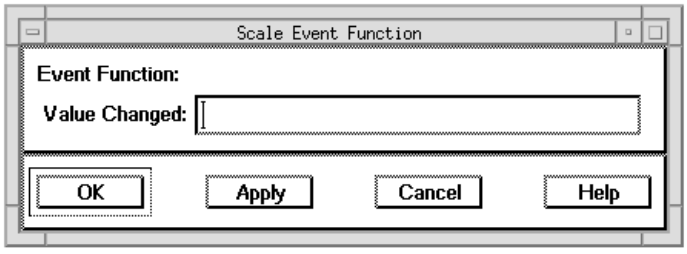
change the maximum value, enter a new value in the entry box.

Minimum	The minimum value of the scale appears in the Minimum entry box. This is the value of the scale if you move the slider all the way to the left. To change the minimum value, type a new value in the entry box.
Width	The width of the scale is displayed (in pixels) in the Width entry box. To change the width of the scale, enter a new value in the Width field.

Scale Events

The Scale Attributes dialog box has optional event function for Value Changed. This allows you to establish a macro to run when the value of the scale is changed from one value to another. To establish a value changed macro, follow these steps:

1. From the Scale attributes dialog box, click **Events**. The Scale Event Function dialog appears.



2. Enter the name of an ELF macro in the Value Changed entry area. Two arguments are passed to this macro: the dialog box handle returned by DB_LOAD@, and the ID of the Scale that received the event.

This feature is useful if you have many scales in an application, and you want a single piece of code to execute whenever one of those scales is changed.

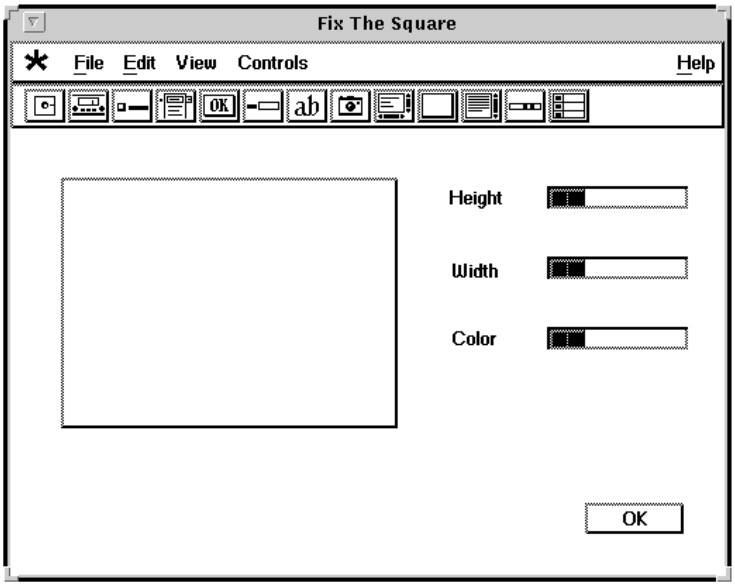
Using DB_CTRL_RETURN_ON_CHANGE@ with Scales

The scale control generates a single exit condition when you change its value. If you include a DB_CTRL_RETURN_ON_CHANGE@ statement in your macro, the scale control generates an exit condition for each unit of change.

For example, suppose you have a scale control with a minimum value of 0 and a maximum value of 100, and you include a DB_CTRL_RETURN_ON_CHANGE@ in your program. If you move the slider from 0 to 50, 50 exit conditions are generated. If you do not have a DB_CTRL_RETURN_ON_CHANGE@ statement, one exit condition is generated.

Scale Example

This example shows a dialog box containing a square panel and three scale controls. The scale controls control the height, width, and color of the square. There is a DB_CTRL_RETURN_ON_CHANGE@ statement for the height control. The dialog box and code follow.




```
MACRO Square

    VAR        dbox,exit_cond,ctrl_value,value_array

'load specified dialog box into memory

    dbox = DB_LOAD@("Square")
    DB_CTRL_RETURN_ON_CHANGE@(dbox, "Height Scale", TRUE)

'initialization section for controls before being displayed

    value_array[0] = 1
    value_array[1, 0] = 0
    value_array[1,1] = 0
    value_array[1,2] = 0
    value_array[2] = "Bob"
    DB_CTRL_WIDGET_COLOR@(dbox, "Panel", value_array)

'
' Sets the Square color to black. If you slide the color scale, the square becomes more
' and more red in color.
'
    WHILE exit_cond <> "OK"
```

```
        DB_DISPLAY@(dbox)

'get the (optional id) for control causing exit condition

        exit_cond = DB_EXIT_CTRL@(dbox)

'process accordingly based on exit condition
        CASE of exit_cond

        CASE "Width Scale"
                DB_CTRL_WIDTH@(dbox,
                        "Panel", DB_CTRL_GET_VALUE@(dbox, "Width Scale"))

        CASE "Height Scale"
                DB_CTRL_HEIGHT@(dbox,
                        "Panel", DB_CTRL_GET_VALUE@(dbox, "Height Scale"))

        CASE "Color Scale"
                value_array[1,0] = DB_CTRL_GET_VALUE@(dbox, "Color Scale")
                DB_CTRL_WIDGET_COLOR@(dbox, "Panel", value_array)

        ENDCASE
  WEND
ENDMACRO
```

Related Macros

The following macros are useful for programming the scale control.

- DB_CTRL_MAX_VALUE@ sets the minimum and maximum value of the scale.
- DB_CTRL_VALUE@ sets the value of the scale.
- DB_CTRL_GET_VALUE@ returns the number of the current slider position.

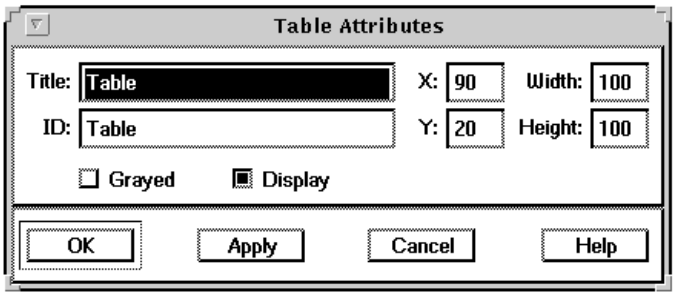
Tables

A table is a control that allows you to display data in columns. A table control looks like this:

To add a table to the dialog box, choose Controls → Table and click in the dialog box. The table appears in the dialog box. The following section describes how to manipulate table attributes.

Table Attributes

When you double-click on a table, the Table Attributes dialog appears:



The Title, ID, Display, X, and Y attributes are common to all ELF controls. These are described in the section "Introduction to ELF Controls", earlier in this chapter. In addition, the Table Attributes dialog allows you to set the following attributes:

Grayed	Turn on Grayed to make the table grayed. Turn off Grayed to make the table normal.
Height	The height of the table in pixels is displayed in the Height entry box. Enter a new value in the entry box to change the height of the table.
Width	The width of the table in pixels is displayed in the Width entry box. Enter a new value in the entry box to change the width of the table.

Initializing Tables

Almost all aspects of a table's appearance, such as the number of columns, the width of the columns, horizontal and vertical rules, and so on, are established programmatically. To display information in a table control, you must perform the following steps:

1. Set up a two-dimensional array that contains your table data. For example:

```
players[0]  =  "Mo",    "Granite", "11/09/66",  "6"
players[1]  =  "Fred",  "Flintstone", "01/20/63", "8"
players[2]  =  "Barney", "Rubble", "12/03/68",  "5"
```

   This example shows data that is displayed in three rows in the table. The first index of the array is the row number. The second index of the array is the column number. Therefore, the third column of the first row of the table will contain the date 11/09/66.

2. Set up a second array that contains your table headings. This 2-dimensional array should have one entry for each column of your data.

```
heading[0] =    "First Name",          130
heading[1] =    "Last Name",           130
heading[2] =    "Date Born",           130
heading[3] =    "Years Played",        300
```

   For each entry in the array, the first index defines the text that appears at the top of the column. The second index defines the width of the column in pixels.

3. Call a series of macros to change the appearance of the table. For example, there are macros that control whether your table has horizontal and vertical lines. These macros are described in the section "ELF Table Macros", later in this chapter.

4. Call DB_PAINT@(dbox, TRUE) to set the table to read-only. Unlike other controls in Applixware, tables must be displayed *before* their data is loaded. Therefore, you must call DB_PAINT@ to prevent users from clicking the table control until initialization is complete.

5. Call DB_TABLE_SET_DATA@ to load the arrays into the table control. The format for this macro is as follows:

```
DB_TABLE_SET_DATA@(dbox, "Table", rows,
                   headings, pixmaps)
```

6. Call DB_CTRL_RETURN_ON_CHANGE@ to establish an exit condition when the user clicks an entry in the table.

7. Call DB_DISPLAY_ONLY@(dbox, FALSE) to allow user input into the dialog box.

ELF Table Macros

The following macros are useful if you are using tables in your dialog box programs:

- DB_TABLE_HIDE_VERTICAL_GRID@ toggles the vertical lines in a table on and off.
- DB_TABLE_HIDE_HORIZONTAL_GRID@ toggles the horizontal lines in a table on and off.
- DB_CTRL_HORIZ_SCROLL@ toggles the horizontal scroll bar in a table on and off.
- DB_CTRL_MULTI_SELECT@ allows the user to select more than one entry in a table.
- DB_DISPLAY_ONLY@ sets the dialog box to Read-only. This is necessary to prevent user input while the table control is being initialized.
- DB_TABLE_SET_DATA@ loads table data, headings and pixmaps into the table control. Pixmaps are described in more detail in the following section "Table Markers".
- DB_TABLE_SET_FONT@ sets the text font for the table. This font applies to both the headers and the body text of the table.

Table Markers

Table Markers are used to mark a table row. For example, if you were developing a mail application, you might want to mark unread mail messages with check marks. The example code later in this section, illustrates this.

To incorporate table markers into your table, follow these steps:

1. Call DB_TABLE_SET_MARKER_WIDTH@ to establish an area at the beginning of each row for the table marker. This is not a table column - it is only an area in which you can display bitmaps.

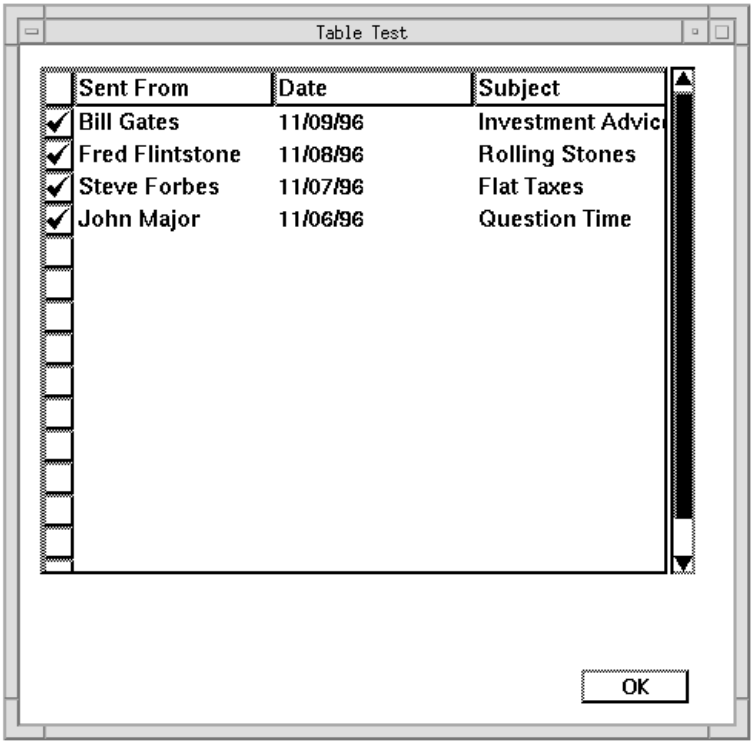
2. Load an array with bitmap images. These images can be generated through the Bitmap Editor, which is described in Chapter 11, "The Bitmap Editor". You can also use the standard Applixware bitmaps that are loaded when Applixware starts.

   The file datafile.dat contains a list of bitmaps (.im files) that are read into memory when Applixware runs. You can use these as table markers. The file is located in axdata/*lang*.

3. Call DB_TABLE_SET_DATA@, and include the array of pixmaps.

Table Example

This example shows a dialog box containing a table. You can select an item in the table by clicking on it. The dialog box and code follow.




```
macro tabletest

var dbox, exit_cond, heading, j, pixmaps, senders, sender_chosen, sender

  dbox = DB_LOAD@("Table")

/*
* Initialize Table Control
*/

  senders[0]          =          "Bill Gates",          "11/09/96", "Investment Advice"
  senders[1]          =          "Fred Flintstone",     "11/08/96",          "Rolling Stones"
  senders[2]          =          "Steve Forbes",        "11/07/96",          "Flat Taxes"
  senders[3]          =          "John Major",          "11/06/96",          "Question Time"

  heading[0] =          "Sent From",                   130
  heading[1] =          "Date",                        130
  heading[2] =          "Subject",                     130
```

```
  DB_CTRL_RETURN_ON_CHANGE@(dbox, "Table", TRUE)
  DB_TABLE_HIDE_VERTICAL_GRID@(dbox, "Table", TRUE)
  DB_TABLE_HIDE_HORIZONTAL_GRID@(dbox, "Table", TRUE)
  DB_CTRL_HORIZ_SCROLL@(dbox, "Table", FALSE)
  DB_TABLE_SET_MARKER_WIDTH@(dbox, "Table", 20)
  DB_CTRL_MULTI_SELECT@(dbox, "Table", FALSE)
  DB_PAINT@(dbox, TRUE)     'needed to set font and data into table
/*
* The marker pixmaps argument is set to be the "check" icon taken from
* the datafile.dat file located in axdata/eng directory. When Applixware starts,
* it reads datafile.dat from the disk. The strings defined in datafile.dat can be used
* as bitmaps. The check icon shows up on each row of the table.
* Note that the pixmaps argument must be defined as a two-dimensional array of
* pixmaps.
*/

  FOR j = 0 TO (ARRAY_SIZE@(senders)-1)
            pixmaps[j,0] = "check"
  NEXT j

  DB_TABLE_SET_FONT@(dbox, "Table", "Helvetica", 14, TRUE, FALSE)
  DB_TABLE_SET_DATA@(dbox, "Table", senders, heading, pixmaps)
  DB_TABLE_ALLOW_EDITING@(dbox, "Table", FALSE)

while exit_cond <> "OK"

            DB_DISPLAY@(dbox)
            exit_cond = DB_EXIT_CTRL@(dbox)
/*
* The table control is handled by this IF statement.
*/

            IF exit_cond = "Table"
            {
            sender_chosen = DB_TABLE_GET_SELECTIONS@(dbox, "Table")
            sender = sender_chosen[0]
            INFO_MESSAGE@(senders[sender,0] ++
                        " sent you a message on "++senders[sender,1]++
                        " regarding "++senders[sender,2])
            }

WEND

endmacro
```

Related Macros

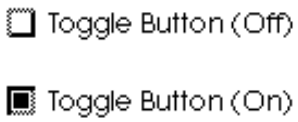
The following macros are useful for programming the table control.

- DB_TABLE_GET_SELECTIONS@ returns an array containing the currently selected table row.
- DB_TABLE_SET_SELECTIONS@ selects rows in a table.
- DB_TABLE_GET_DATA@ returns an array of string containing all the data in the table.
- DB_TABLE_SET_DATA@ adds strings and markers to a table.
- DB_TABLE_ALLOW_EDITING@ allows users to type information into the table. A table is either editable or selectable. If a table is selectable, you cannot edit it. If a table is editable, you cannot select rows by clicking on them with the mouse.

  DB_TABLE_ALLOW_EDITING@ allows you to toggle between an editable table (editing is allowed) and a selectable table (editing is not allowed.)

Toggle Buttons

A toggle button is a button that has two states: on or off. A toggle button in an ELF dialog box looks like this:

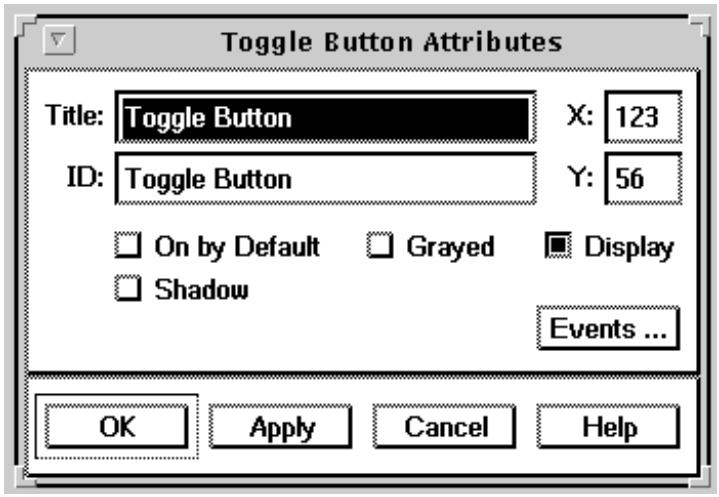


To add a toggle button to the dialog box, choose Controls → Toggle Button and click in the dialog box. The toggle button appears in the

dialog box. The following section describes how to manipulate toggle button attributes.

Toggle Button Attributes

When you double-click a toggle button in a dialog box, the Toggle Button Attributes dialog appears:



The Title, ID, Display, X, and Y attributes are common to all ELF controls. These are described in the section "Introduction to ELF Controls", earlier in this chapter. In addition, the Toggle Button Attributes dialog box allows you to set the following attributes:

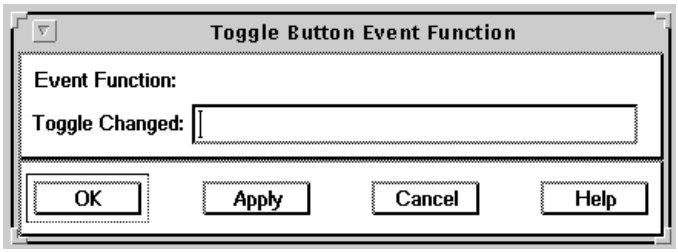
Events	Toggle button events are discussed in the next section.
Grayed	Turn on Grayed to make the toggle button grayed. Turn off Grayed to make the toggle button normal.

On by Default	Turn on On by Default to make the toggle button on when it appears. Turn off On by Default to make the toggle button off when it appears.
Shadow	Turn on Shadow to make the toggle button display with a shadow attribute. Turn off Shadow to make the toggle button display normal.

Toggle Changed Event

The toggle button has an optional event function. This allows you to establish a macro that runs whenever the toggle button is clicked. To establish a Toggle Changed event macro, follow these steps:

1. From the Toggle Button Attributes dialog box, click **Events**. The Toggle Button Event Function dialog box appears.



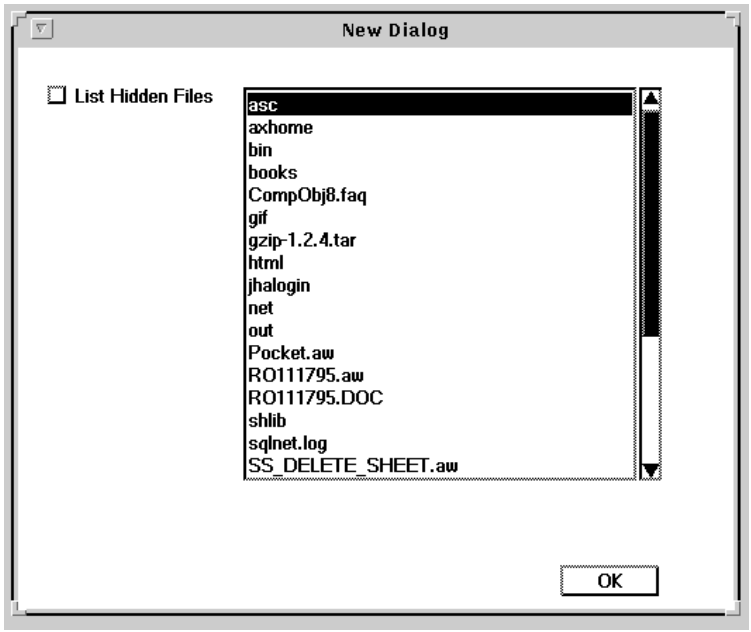
2. Enter the name of an ELF macro in the Toggle Changed entry box. This macro runs whenever you click the toggle button. Two arguments are passed to this macro: the dialog box handle returned by DB_LOAD@, and the ID of the toggle button that received the event.

This feature is useful if you have many toggle buttons in an application, and you want a single piece of code to execute whenever one of those button is clicked. The processing of this macro supercedes the execution of the exit condition loop, as illustrated below:

Macro Code	Event Code
MACRO ToggleEventTest /* * 1. Declare Variables * 2. Load the dialog box * 3. Establish exit conditions. * The While loop below is used to * process exit conditions. Because the * toggle button has the ToggleChanged * event defined, the tests for the Toggle * Button exit condition are never executed. */ while exit_cond <> "OK" DB_DISPLAY@(dbox) exit_cond = DB_EXIT_CTRL@(dbox) if exit_cond = "Toggle Button" { **/*** *** This code is superceded by the** *** ToggleEvent Macro.** ***/** } WEND ENDMACRO	**/*** *** This code is executed when the toggle** *** button is clicked.** ***/** MACRO ToggleEvent(dbox, ToggleID) info_message@("You clicked the Togglebutton!") ENDMACRO

Toggle Button Example

The following example lists the files in your $AXHOME directory. If you press the toggle button, ToggleTest displays the hidden files in your $AXHOME directory. The dialog box and code follow.




```
MACRO ToggleTest

/*
*      Declare Variables
*/
var dbox, exit_cond, array1, array2,  arg, SortArray1, sortarray2

/*
*      Load the dialog box and establish exit conditions.
*/
       dbox = DB_LOAD@("toggletest.d")
       DB_CTRL_RETURN_ON_CHANGE@(dbox,"Toggle Button",TRUE)

/*
*      Establish 2 arrays. SortArray1 is displayed when the toggle
*      button is off. SortArray2 is displayed when the toggle button
*      is on. SortArray1 is loaded as the default.
*/
       array1 = SHELL_COMMAND@("ls $HOME")
       array2 = SHELL_COMMAND@("ls -a $HOME")
```

```
        SortArray1 = SORT@(array1, FALSE)
        SortArray2 = SORT@(array2, FALSE)
        arg = DB_CTRL_STRINGS@(dbox, "List Box", SortArray1)

/*
*       The While loop below is used to process exit conditions
*       from the dialog box.  Only two exit conditions are defined:
*       1. The user presses OK.
*       2. The user presses the List Hidden Files Toggle Button.
*/

while exit_cond <> "OK"

        DB_DISPLAY@(dbox)
        exit_cond = DB_EXIT_CTRL@(dbox)

        if exit_cond = "Toggle Button" and
                DB_CTRL_GET_VALUE@(dbox, "Toggle Button") = TRUE
        {
        arg = DB_CTRL_STRINGS@(dbox, "List Box", SortArray2)
        }

        if exit_cond = "Toggle Button" and
                DB_CTRL_GET_VALUE@(dbox, "Toggle Button") = FALSE
        {
        arg = DB_CTRL_STRINGS@(dbox, "List Box", SortArray1)
        }

WEND

ENDMACRO
```

Related Macros

The following macros are useful when programming toggle buttons:

- DB_CTRL_VALUE@(dbox, "Toggle Button", value) sets the toggle button on (value = 1), off(value = 0) or grayed(value = 2).
- DB_CTRL_GET_VALUE@(dbox, "Toggle Button") gets state of the toggle button. If this macro returns TRUE(-1) the toggle button is on. If this macro returns FALSE (0) the toggle button is off.

9 Dialog Box Programming

This chapter covers the following topics:

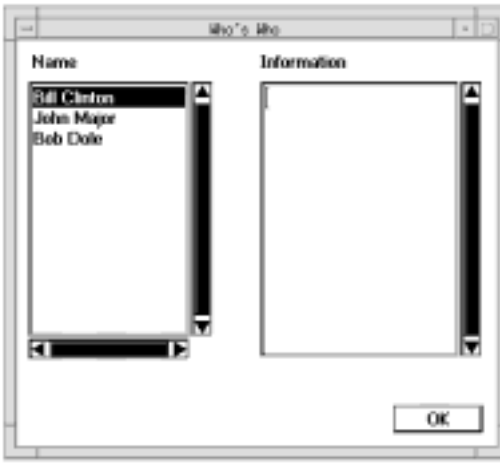
- Dialog box program structure
- Disabling and hiding controls
- Positioning a dialog box on the screen

Dialog Box Program Structure

Once you have created a dialog box using the dialog box editor, you can include it in a macro. All ELF macros that use dialog boxes have the same basic structure, with five parts, as follows:

- Part 1 - Define variables for use in the ELF macro
- Part 2 - Load the dialog box from the disk with DB_LOAD@.
- Part 3 - Define *exit conditions* based on the controls in the dialog box. Exit conditions are user events that suspend operation of the dialog box task, and return control to the ELF macro. Two macros are primarily used to define exit conditions: DB_CTRL_RETURN_ON_CHANGE@ and DB_CTRL_TYPING_RETURN@.
- Part 4 - Set up data to be displayed in the dialog box and load the information into the controls.
- Part 5 - A loop that displays the dialog box, and processes user-driven events based on the exit conditions resulting from those events.

The Structure macro that follows shows the typical structure of a dialog box macro.



MACRO Structure

```
/*
*       Part 1 - Define Variables
*/

VAR dbox,exit_cond
VAR ListBoxArray, Clinton_String,
     Major_String, Dole_String

/*
*    Part 2 - Load the Dialog Box from disk and store it in a variable named dbox.
*/
     dbox = DB_LOAD@("Structure.d")

/*
*    Part 3 - Establish Exit Conditions. Exit conditions indicate that control will be
*    passed from the dialog box back to the ELF macro when an action is performed
*    using that control in the dialog box.  For example, when the user clicks on an
*    entry in the list box, control returns from the dialog box back to the macro.
*/

     DB_CTRL_RETURN_ON_CHANGE@(dbox,"List Box",TRUE)

/*
*    Part 4 - Initialize controls before display
*/

     ListBoxArray = "Bill Clinton", "John Major", "Bob Dole"
     Clinton_String = "President of the United States"
     Major_string = "Prime Minister of Great Britain"
     Dole_String = "Senate Majority Leader"
     DB_CTRL_STRINGS@(dbox, "List Box", ListBoxArray)
     done = FALSE

/*
*    Part 5 - This WHILE loop processes user events based on
*    exit conditions
```

```
*/

    WHILE exit_cond <> "OK"

            DB_DISPLAY@(dbox) 'display dialog box to screen

            exit_cond = DB_EXIT_CTRL@(dbox)

/*
*   DB_EXIT_CTRL@ gets the ID for the control causing the exit condition. For
*   example, determine what information was changed, or what button was pushed.
*/

' The exit conditions are processed in this CASE statement
    CASE of exit_cond

            CASE "List Box"     'User clicked a list box entry
            If DB_CTRL_GET_VALUE@(dbox, "List Box") = 0 then
                    DB_EDITBOX_SET_DATA@(dbox,"Editbox", Clinton_String)

            If DB_CTRL_GET_VALUE@(dbox, "List Box") = 1 then
                    DB_EDITBOX_SET_DATA@(dbox,"Editbox", Major_String)

            If DB_CTRL_GET_VALUE@(dbox, "List Box") = 2 then
                    DB_EDITBOX_SET_DATA@(dbox,"Editbox", Dole_String)

/*
*           CASE "Another Control"   ' Dialog Boxes boxes typically have multiple
*                                       ' controls.  The CASE statement has one case
*                                       ' for each control that receives user events.
*/
    ENDCASE

    WEND

ENDMACRO
```

The following five sections describe each part of this program in detail.

Defining Variables

All ELF variables are defined by the VAR statement. In a typical dialog box macro, the defined variables include:

- One for the exit condition returned from the dialog box. This variable is loaded using the DB_EXIT_CTRL@ macro.
- One for the dialog box itself. This variable is loaded using the DB_LOAD@ macro.

Other variables are defined as required.

Loading the Dialog Box from Disk

Use the DB_LOAD@ macro to read the dialog box file from the disk into memory. The DB_LOAD@ macro looks in the directories specified in the elf_search_list@ system variable. If the dialog box is not in one of those directories, DB_LOAD@ looks in the */install_dir*/axdata directory.

Defining Exit Conditions

Exit conditions are user events that pass control of processing from the dialog box to your ELF macro. For example, if a user changes the value of an option button, or selects a new entry in a list box, you can program your macro to return processing control to your ELF code so that you can respond to the user event.

Three macros are used to establish exit conditions for controls:

- DB_CTRL_RETURN_ON_CHANGE@
- DB_CTRL_TYPING_RETURN@
- DB_CTRL_ACTIVE_RETURN@

All three of these macros are discussed in the sections that follow.

DB_CTRL_RETURN_ON_CHANGE@

The following table shows the controls for which DB_CTRL_RETURN_ON_CHANGE@ is relevant, and the user event that triggers the exit condition.

Table 9-1 DB_CTRL_RETURN_ON_CHANGE@ Events

Control	User Event
Scale	Drag the scale slider to a new value. With DB_CTRL_RETURN_ON_CHANGE@ in effect, an exit condition is generated for each unit of change. If you move the slider from 0 to 100, 100 exit conditions are generated.
List Box	Click an item in the List Box
Radio Button Group	Change Radio Button value
Toggle Button	Change Toggle Button value
Push Button	Click the button
Option Button	Change Option Button value
Table	Change the selected row. The exit condition is triggered as long as the table is not editable. For more information, see the section on "Tables" in Chapter 8, "ELF Control Reference".

DB_CTRL_TYPING_RETURN@

This macro generates an exit condition when you type a character in an Edit Box, Entry Box, or Table. For more information refer to Chapter 8, "ELF Control Reference".

DB_CTRL_ACTIVE_RETURN@

This macro generates an exit condition when the edit box or entry box gains *focus*. A control has focus when user input, such as typing at the keyboard, is directed to that control. Typically, a control gains focus when you click on it with the mouse.

Initializing Controls

Before you display the dialog box on the screen, you must initialize each control, by adding data to it or changing its appearance.

Initializing controls can sometimes require several macros, depending on the type of control you are using. Table 9-2 shows how to initialize each ELF control. For more information refer to Chapter 8, "ELF Control Reference".

Table 9-2 Initializing Controls

Control	Initialization
Scale	The maximum and minimum values of a scale are established when you create the scale in the dialog box. You can set these values through the Scale Attributes dialog. You can also set the maximum and minimum value through the macros DB_CTRL_MAX_VALUE@ and DB_CTRL_MIN_VALUE@. The initial value of the scale is set through the macro DB_CTRL_VALUE@.
List Box	DB_CTRL_STRINGS@ adds strings to the list box. DB_CTRL_VALUE@ specifies the option selected.
Radio Button Group	Strings for each selection are added at design time through the Radio Button Attributes dialog. You can also add options to the Radio Button with the macro DB_CTRL_STRINGS@. Use DB_CTRL_VALUE@ to specify the option selected.
Toggle Button	Strings for each selection are added at design time through the Toggle Button Attributes dialog. DB_CTRL_VALUE@ specifies the option selected.
Push Button	The title of the Push button is established at design time through the Push Button Attributes dialog.
Option Button	DB_CTRL_STRINGS@ sets the option labels. DB_CTRL_VALUE@ specifies the option selected.
Table	DB_TABLE_SET_DATA@ adds strings and markers to the table. Use DB_TABLE_ALLOW_EDITING@ to allow users to type information into the table.
Edit Box	DB_EDITBOX_SET_DATA@ sets text in the edit box.
Entry Box	DB_CTRL_VALUE@ sets the string in the Entry Box. You can set the cursor in the entry box, and highlights characters in the entry box with the macro DB_CURSOR_IN_ENTRY@ .

Uninitialized Controls

If you do not initialize control values, they are displayed as follows:

- Entry boxes appear blank
- Toggle buttons are not selected
- Tables are a blank square with no grid lines
- For radio button groups, option buttons, and list boxes, the first item is selected.

Occasionally, you might not want any items to be selected in a list box, radio button group, or option button. Use DB_CTRL_VALUE@ to set the value for the control to -1. This specifies that no items be selected in these controls.

To make a toggle button blank (neither toggled on or toggled off), set the value for the button control to 2. The Toggle button appears a grayed dot.

Processing Exit Conditions

The final part of a dialog box macro is typically handled by a loop combined with a series of conditional statements, such as a set of IF statements or a CASE statement. The Loop in the Structure macro follows:

```
WHILE exit_cond <> "OK"

    DB_DISPLAY@(dbox)
    exit_cond = DB_EXIT_CTRL@(dbox)

    CASE of exit_cond

        CASE "List Box"
        If DB_CTRL_GET_VALUE@(dbox, "List Box") = 0 then
            DB_EDITBOX_SET_DATA@(dbox,"Editbox", Clinton_String)
```

```
            If DB_CTRL_GET_VALUE@(dbox, "List Box") = 1 then
                    DB_EDITBOX_SET_DATA@(dbox,"Editbox", Major_String)

            If DB_CTRL_GET_VALUE@(dbox, "List Box") = 2 then
                    DB_EDITBOX_SET_DATA@(dbox,"Editbox", Dole_String)

        ENDCASE

    WEND
```

This loop works as follows:

1. Test for an exit condition of "OK". This exit condition occurs when the user presses the OK button.
2. Display the dialog box. The most commonly-used macro for diaplying a dialog box is DB_DISPLAY@.
3. Use DB_EXIT_CTRL@ to determine which control in the dialog box triggered the exit condition. For example, if the user pressed the OK button, the exit condition would be the string "OK".
4. Use an IF statement or CASE statement to respond to different user events.
5. Re-display the dialog box and wait for more user events.

The next five sections describe these steps.

WHILE exit_cond <> "OK"

This is the first line of the main loop of the Structure program. It tests the dialog box exit condition to see if the user pressed the OK button. The **OK** button is a button of type Execute & Dismiss. These buttons automatically trigger an exit condition when they are selected.

NOTE: Buttons of type Execute & Dismiss do not require an explicit DB_CTRL_RETURN_ON_CHANGE@ to establish an exit condition.

Most dialog boxes you create should include a Dismiss button for exiting the dialog box. In order to cancel a dialog box, you need to include the following statement in your macro:

```
IF DB_CANCELLED@(dbox)
    RETURN
```

DB_CANCELLED@ checks whether a Dismiss button has been clicked. If a Dismiss button has been selected, RETURN is executed to exit the dialog box. DB_CANCELLED@ takes the name of your dialog box variable as an argument.

Pressing a Dismiss button should exit a dialog box without performing any action or saving any control changes that have been made. The DB_CANCELLED@ statement should be included in your dialog box macro before any statements that may perform an action or save control settings. For example, you may want to include DB_CANCELLED@ immediately after you display the dialog box:

```
MACRO ExampleDlg
    VAR dbox
    dbox = DB_LOAD@("dbox_file")
    DB_DISPLAY@(dbox)
    IF DB_CANCELLED@(dbox)
        RETURN
ENDMACRO
```

DB_DISPLAY@

After loading the dialog box file with DB_LOAD@, you display the dialog box using the DB_DISPLAY@ macro. The DB_DISPLAY@ macro takes the dialog box variable as an argument.

While running, DB_DISPLAY@ changes the value of the passed dialog box variable according to changes made in the dialog box. Usually, the DB_DISPLAY@ macro remains active until the dialog box is exited.

DB_EXIT_CTRL@

Each control has a Control ID attribute. The Control ID attribute is a string that is passed from the dialog box to the ELF macro when a user event occurs. By examining the returned string with the macro DB_EXIT_CTRL@, you can determine which control in the dialog box triggered the exit condition. The format of DB_EXIT_CTRL@ is:

```
DB_EXIT_CTRL@(dbox)
```

Because the Control attributes are used by ELF to determine the user event, the Control IDs must be unique within the dialog box.

The argument dbox is the name of the dialog box variable for the dialog box in which you want to identify the control responsible for the exit condition.

Retrieving information from Dialog Box Controls

Once you have determined what control the user acted on, you might want to read information from that control. Table 9-3 shows the ELF macros that you use to extract information from controls.

Table 9-3 Reading Information from Controls

Control	Macros
Scale	DB_CTRL_GET_VALUE@ returns the number of the current slider position.
List Box	DB_CTRL_GET_VALUE@ returns the number of the selected option. DB_CTRL_GET_STRINGS@ returns an array of options that are displayed in the list box.
Radio Button Group	DB_CTRL_GET_VALUE@ returns the number of the selected option. DB_CTRL_GET_STRINGS@ returns an array of options that are displayed in the radio button.
Toggle Button	DB_CTRL_GET_VALUE@ returns TRUE if the toggle button is on, and FALSE if the toggle button if off.
Option Button	DB_CTRL_GET_VALUE@ returns the number of the selected option. DB_CTRL_GET_STRINGS@ returns an array of options that are displayed in the option button.
Table	DB_TABLE_GET_SELECTIONS@ returns an array containing the currently selected table row. DB_TABLE_GET_DATA@ returns an array of strings containing all the data in the table.
Edit Box	DB_EDITBOX_GET_DATA@ returns the contents of the edit box.
Entry Box	DB_CTRL_GET_VALUE@ returns the string entered in the entry box.

Disabling and Hiding Controls

You can disable and hide controls in a dialog box in two ways:

- Through control attributes at design time. Each dialog box control has a Display toggle button and a Grayed toggle button in its attributes dialog. Disabling a control is called *graying* because the control is displayed in gray shades on the screen. You can change these settings by double-clicking the control in the dialog box editor.
- Through ELF macros at run time. The ELF macros that gray and hide controls are DB_CTRL_GRAYED@ and DB_CTRL_DISPLAY@.

Disabling Controls

You can disable a control so that it cannot be selected using the DB_CTRL_GRAYED@ macro. The format for the macro is as follows:

```
DB_CTRL_GRAYED@(dbox, control_ID, value)
```

The value argument can be TRUE or FALSE: TRUE disables a control, FALSE enables the control.

To disable a control when the dialog box is first displayed, place the DB_CTRL_GRAYED@ macro after the DB_LOAD@ macro and before the DB_DISPLAY@ macro.

Disabling List Boxes

When you disable a list box, you must add code to prevent users from selecting items in the list box. If you do not, the items in the list box can still be selected, even though the control is disabled.

The code fragment that follows deselects all the items in a list box whenever the user clicks it.

```
again:
    DB_DISPLAY@(dbox)
    exit_cond = DB_EXIT_CTRL@(dbox)
    IF (exit_cond = "List Box" ) AND
        (DB_CTRL_GRAYED@(dbox,"List box",TRUE))
    {
        DB_CTRL_VALUE@(dbox, "List box", -1)
        GOTO again
    }
```

Hiding Controls

You can hide controls in a dialog box so they do not appear until you want them to. You can turn off the Display toggle button in the Control Attributes dialog box, or you can hide a control using the DB_CTRL_DISPLAY@ macro. The format for DB_CTRL_DISPLAY@ is as follows:

```
DB_CTRL_DISPLAY@(dbox, control_ID, value)
```

The value argument can be TRUE or FALSE: TRUE means the control is displayed, FALSE means the control is hidden.

To make a control hidden when the dialog box is first displayed, place the DB_CTRL_DISPLAY@ macro before the DB_DISPLAY@ macro in your dialog box macro.

Positioning a Dialog Box on the Screen

Usually, when a dialog box is displayed, it is automatically centered under the current mouse pointer location. If you want, you can use the DB_XPOS@ and DB_YPOS@ macros to specify where a dialog box should be placed on the screen when it is displayed.

The formats for the macros are as follows:

```
DB_XPOS@(dbox,value)
DB_YPOS@(dbox,value)
```

The value is a number indicating the location, in pixels, of the top left corner of the dialog box. The location is with respect to the top left corner of the screen (position (0,0)). The value of DB_XPOS@ indicates the horizontal position of the dialog box. The value of DB_YPOS@ indicates the vertical position of the dialog box.

The DB_XPOS@ and DB_YPOS@ macros must appear before the DB_DISPLAY@ macro.

For example, the following macro positions the dialog box at location (150,250) on the screen.

```
MACRO DboxMacroDlg
    VAR dbox

    dbox = DB_LOAD@("dbox_file")
    DB_XPOS@(dbox,150)
    DB_YPOS@(dbox,250)
    DB_DISPLAY@(dbox)
    IF DB_CANCELLED@(dbox)
        RETURN
ENDMACRO
```

If you set the values for both DB_XPOS@ and DB_YPOS@ to 0, the dialog box is centered under the current mouse pointer location. Therefore, the closest you can position a dialog box to the top left corner of the screen is position (1,0) or position (0,1).

If you use DB_XPOS@ to set the horizontal position of the dialog box without using DB_YPOS@ to set the vertical position, then the vertical position is set to 0. Likewise, if you set the vertical position using DB_YPOS@ but do not set the horizontal position, the horizontal position is set to 0.

The maximum values for DB_XPOS@ and DB_YPOS@ depend on the dimensions of your screen.

10 Advanced Dialog Box Programming

This chapter describes enhancements that you can make to ELF programs that use dialog boxes. This chapter covers the following topics:

- Using menu bars in dialog boxes
- Maintaining a dialog box display
- Displaying multiple dialog boxes
- Pokes
- Using macros to create or change dialog boxes

Using Menu Bars in Dialog Boxes

A dialog box can include a menu bar of pull-down menus similar to the menus found in the Applixware applications. To create a menu bar in a dialog box, follow these steps:

1. Define the menu bar. There are two ways to define a menu bar:
   - Use the Menu Bar Editor to define a menu bar, and save the menu bar to a file.
   - Write a macro to define a menu bar array, and write the array to a file using the macro WRITE_DATA_FILE@.
2. Use the macro READ_DATA_FILE@ to read the menu bar file from the file. READ DATA FILE@ returns an array containing all of the information in the file.
3. Use SET_SELECTIONS@ to assign a menu bar ID to the array that you read from the file in step 2.
4. Run the macro DB_MENU_BAR@ to associate the menu bar with your dialog box.
5. Write code in your ELF macro to handle menu bar events.

These steps are described in the following sections.

Defining a Menu Bar

You can use the Menu Bar Editor to create a new menu bar, or you can make a copy of an existing menu bar file and use the Menu Bar Editor to change the copy to suit your needs.

All Applixware menu bar definition files are saved in *install path*/axdata/*eng*, where install path is the directory in which you installed Applixware, and eng is language-dependent. For example, in a french translation of Applixware, this directory would be *install path*/axdata/*frn*.

When you customize your menu bar, Applixware places a customized copy of the file in your axhome directory. Some of the Applixware menu bar definition files are listed in Table 10-1.

Table 10-1 Menu Bar Definition Files

Menu Bar File	Applixware Application
ax_wp4	Applix Words
ax_de4	Dialog Box Editor
ax_ss4	Applix Spreadsheets
ax_mm4	Applixware Iconbar
ax_gr4	Applix Graphics
ax_sq4	Applix Data
ax_me4	Macro Editor

You can copy one of these files and edit it to create a menu bar for your dialog box. However, Applix recommends that you create menu bars for your application by running the macro MENU_BAR_DLG@.

```
MENU_BAR_DLG@(ID_number, name)
```

ID_number is a unique number that you specify as the identification for the menu bar definition. The number must be in the range from 20 to 100.

name is the name of the menu bar file. If you create a new file, supply a new, unique name for the file. All menu bar definition files are

automatically saved in the user's home directory, so do not supply a path name or a file extension.

For a description of how to use the Menu Bar Editor, see *Getting Started*.

Reading the Menu Bar File from a File

You read a menu bar file using the READ_DATA_FILE@ macro. The format for the macro is:

```
READ_DATA_FILE@(filename)
```

The filename is a string containing the full path name of the menu bar file. Since menu bar files reside in your home directory, the path for filename should be your home directory.

READ_DATA_FILE@ returns an array containing the data from the file. You assign the result of READ_DATA_FILE@ to a variable. For example, if you declare the variable result and the pathname of the menu definition file is /user/sam/axhome/my_bar, you include the following statement to load the contents of the data file:

```
result = READ_DATA_FILE@("/user/sam/axhome/my_bar")
```

You can also use the AXHOME_DIR@ macro to automatically set the path to your customized menu bar:

```
result = READ_DATA_FILE@(AXHOME_DIR@++"/my_bar")
```

Assigning a Menu Bar ID to the Array

Use SET_SELECTIONS@ to assign a menu bar ID to the array returned from READ_DATA_FILE@. The format of the macro is:

```
SET_SELECTIONS@(ID_number, menu_array)
```

ID_number is a unique number that you specify as the identification for the menu bar definition. The number must be in the range from 20 to 100. If you created the menu bar using the Menu Bar Editor, ID_number should be the same as the ID number you provided with the MENU_BAR_DLG@ macro.

menu_array is the name of the array that contains your menu bar definition data.

For example, the following macro assigns the identification number 23 to the menu bar array my_bar.

```
SET_SELECTIONS@(23,my_bar)
```

SET_SELECTIONS@ must be placed in your dialog box macro before DB_DISPLAY@ and after DB_LOAD@.

Associating the Menu Bar with the Dialog Box

You load the menu bar using the DB_MENU_BAR@ macro. The format for the macro is:

```
DB_MENU_BAR@(dbox, menu_ID)
```

dbox is the name of the dialog box variable.

menu_ID is the ID number for the menu bar as specified with SET_SELECTIONS@ and MENU_BAR_ID@.

DB_MENU_BAR@ must be after DB_LOAD@ and before DB_DISPLAY@ in your dialog box macro.

For example, the following macro loads the menu bar with the ID number 23:

```
DB_MENU_BAR@(dbox, 23)
```

Writing Code to Handle Menu Bar Events

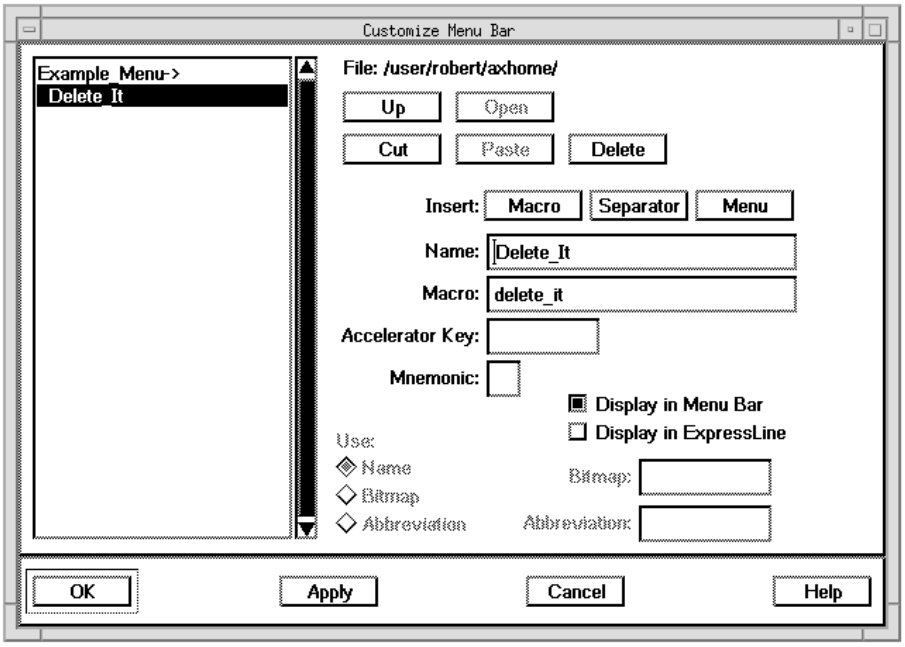
When a menu option is selected, an exit condition is triggered. You use DB_EXIT_CTRL@ to determine if the exit condition was triggered by a control or a menu bar event. If any menu bar option is selected, DB_EXIT_CTRL@ returns menu_bar_.

NOTE: The menu_bar_ exit condition is all lower case, with a trailing underscore.

DB_MENU_BAR_WORD@

Once you've determined that a menu selection triggered an exit condition, use DB_MENU_BAR_WORD@ to determine which menu option was selected. DB_MENU_BAR_WORD@ returns the macro string of your option definition.

For example, suppose you define an option delete_it as shown in the folowing screen:



DB_MENU_BAR_WORD@ returns the string delete_it, which is what is entered in the Macro field.

The format for DB_MENU_BAR_WORD@ is:

```
DB_MENU_BAR_WORD@(dbox)
```

dbox is the name of the dialog box variable.

After determining the option that was selected, you can specify an action to perform based on the selection. You can use a CASE statement or an IF-THEN-ELSE clause for handling menu bar selections. For example:

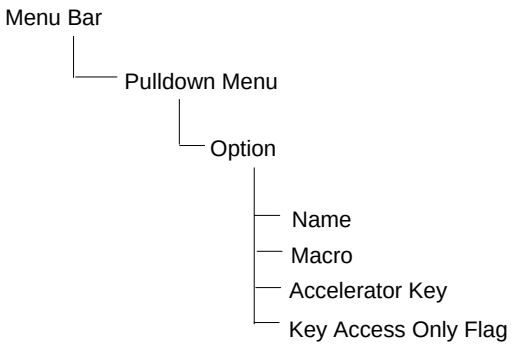
```
exit_cond = DB_EXIT_CTRL@
IF exit_cond = "menu_bar_"
{
```

```
menu_select = DB_MENU_BAR_WORD@(dbox)
IF menu_select = "New"
{
        ' Open a new file
}
ELSE IF menu_select = "Open"
{
        ' Open an existing file
}
ELSE IF menu_select = "Exit"
{
        DB_CANCELLED@(dbox)
}
...
```

Creating a Menu Bar Using an Array

You can define a menu bar using an array definition rather than using the Menu Bar Editor.

A menu bar definition is an array of arrays. The following shows the complete structure of a menu bar array :



The top-level array is the menu bar. The elements in this top-level array are arrays representing the pulldown menus. The pulldown menus consist of arrays that define the options in the pulldown menus. The options in the pulldown menus are arrays that consist of the following elements:

name	The string that appears in the pull-down menu.
macro	The macro that is called when the option is selected. This can either be a built-in ELF macro, or a macro that you write yourself. Note, that choosing the menu item does not call the macro. You must provide a statement in your dialog box macro to call the macro.
accelerator key	A keyboard equivalent for the option. For example, the keyboard equivalent of the Edit → Undo option in an Applix application is F2. Use the standard accelerator key notation. For example, to specify the accelerator key CTRL-B, you would supply the argument ''^B''.
key access only	indicates whether the option should appear on the menu bar or whether it should only be accessible using the accelerator key. If you specify TRUE, the option is only accessible using the accelerator key. If you specify FALSE, the option appears on the menu bar. The default is FALSE.

Menu Bar Array Example

The following example shows a simplified version of the Macro Editor menu bar:

```
xample_Bar
  File
    New
    Open
    Save Formatted
  Edit
    Undo
        Name = Undo
        Macro = WP_UNDO@
        Accelerator Key = !F2
        Key Access Only Flag = FALSE
    Redo
        Name = Redo
        Macro = WP_REDO@
        Accelerator Key = F2
        Key Access Only Flag = FALSE
    Cut
  View
    Expressline
    Ruler
    Boundaries
```

Example_Bar is the name of the top-level array.

File, Edit, and View are the pulldown menus.

All other entries - New, Open, Save Formatted and so on - are option definitions. Option definitions are four element arrays consisting of a name, a macro, an accelerator key, and a Key Access Only flag.

For conciseness, only the elements of the UNDO and REDO options are shown here.

Table 10-2 shows some of the elements in the ExampleBar array. For conciseness, only the elements of the undo and redo object definitions are shown in the table. Other object definitions, such as Cut, would also have four element arrays, but they are not shown in the table.

Table 10-2 Array Elements of Example_Bar

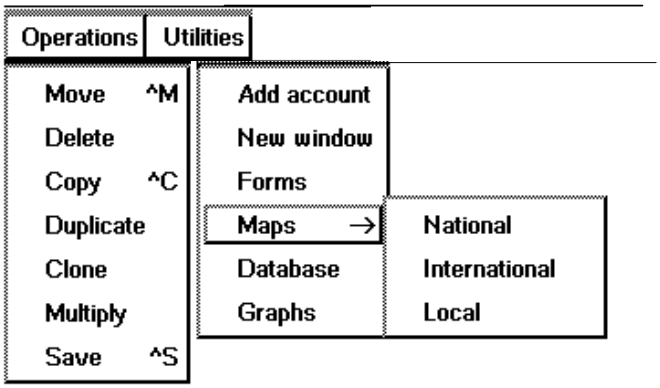
Array Element	Value	Description
Example_Bar[0]	File	A three element array containing the pull-down menu definitions for New, Open, and Save Formatted.
Example_Bar[1]	Edit	A three element array containing the pull-down menu definitions for Undo, Redo and Cut.
Example_Bar[1,0]	Undo	An object definition array containing a Name, a Key, an accelerator key, and a key access only flag.
Example_Bar[1,0,0]	"undo"	A string that appears as a menu element.

Table 10-2 Array Elements of Example_Bar (cont.)

Array Element	Value	Description
Example_Bar[1,0,1]	WP_UNDO@	A macro name. This name is actually the menu word returned by DB_MENU_WORD@.
Example_Bar[1,0,2]	F2	Keyboard notation for the F2 key.
Example_Bar[1,0,3]	FALSE	A boolean value indicating that toggles the Key Access Only flag off.
Example_Bar[1,1]	Redo	An object definition array containing a Name, a Key, an accelerator key, and a key access only flag.
Example_Bar[1,1,0]	"redo"	A string that appears as a menu element.
Example_Bar[1,1,1]	WP_REDO@	A macro name. This name is actually the menu word returned by DB_MENU_WORD@.
Example_Bar[1,1,2]	!F2	Keyboard notation for the Shift and F2 key.
Example_Bar[1,1,3]	FALSE	A boolean value indicating that toggles the Key Access Only flag off.
Example_Bar[1,2]	Cut	An object definition array containing a Name, a Key, an accelerator key, and a key access only flag.
Example_Bar[2]	View	A three element array containing the pull-down menu definitions for Expressline, Ruler, and Boundaries.

Cascading Menus

You can add cascading menus to a menu bar definition array. The following shows an example menu bar that contains two pull-down menus. The Tools pulldown menu includes a cascading menu called Maps.



This menu is defined as follows:

```
VAR menu_bar, operate, options1, options2, type, utiility
'Add operations pull-down menu
options1[0] = "Move", "move_it", "^M",FALSE
options1[1] = "Delete", "delete_it"
options1[2] = "Copy", "copy_it", "^C",FALSE
options1[3] = "Duplicate", "dup_it"
options1[4] = "Clone", "clone_it"
options1[5] = "Multiply", "multiply_it"
options1[6] = "Save", "save_it","^S",FALSE
operate = "Operations", options1
menu_bar[0] = operate
'Add utilities pull-down menu
options2[0] = "Add account", "add"
options2[1] = "New window", "new_win"
options2[2] = "Forms", "form"
'add options to cascading menu
      type[0] = "Local", "local_map"
      type[1] = "National", "nat_map"
      type[2] = "International", "intl_map"
options2[3] = "Maps", type      'cascading menu
```

```
options2[4] = "Database", "data_base"
options2[5] = "Graphs", "graph"
utility = "Utilities", options2
menu_bar[1] = utility
```

Saving the Menu in an ELF Data File

If you have created a separate macro for the menu bar definition, you need to save the definition to an ELF data file using the WRITE_DATA_FILE@ macro. The format for the macro is:

```
WRITE_DATA_FILE@(filename, array_name)
```

filename, a string, is the full pathname of the file you want to write to. If filename is the name of an existing file, the contents of the file will be overwritten.

array_name is the name of the array that you want to write to the data file. For menu bar definitions, array_name is the name of the menu bar array.

Menu Bar Examples

The following macro defines a menu bar having one pull-down menu and writes the definition to the file named /user/joe/menus/utility.mb.

```
MACRO MyMenu
    VAR menu_bar, utility, options, type
    options[0] = "Add account", "add"
    options[1] = "New window", "new_win"
    options[2] = "Forms", "form"
            'add options to cascading menu
            type[0] = "Local", "local_map"
            type[1] = "National", "nat_map"
            type[2] = "International, "intl_map"
    options[3] = "Maps", type
```

```
    options[4] = "Database", "data_base"
    options[5] = "Graphs", "graph"
    utility = "Utilities", options
    menu_bar[0] = utility
    WRITE_DATA_FILE@("/user/joe/menus/utility.mb",menu_bar)
ENDMACRO
```

An Example Menu Bar Defined Using Arrays

The following is an example of a menu bar that includes two pull--down menus. The menu bar is defined in a macro separate from the dialog box macro.

In this example, actions are only defined in the macro for menu options in one of the pull-down menus. To make the other pull-down menu useful, you need to define actions for each of the options in the pull-down menu.

```
MACRO MakeBar
    VAR bar, menu, file, opts

    menu[0] = "Macro", "my_macro"
    menu[1] = "OK", "ok"
    menu[2] = "Not OK", "not_ok"
    file = "File", menu
    bar[0] = file

    menu[0] = "Option1", "opt1"
    menu[1] = "Option2", "opt2"
    menu[2] = "Option3", "opt3"
    menu[3] = "Option4", "opt4"
    opts = "Options", menu
    bar[1] = opts

    WRITE_DATA_FILE@("/user/joe/my_menubar",bar)
ENDMACRO

MACRO DbBar
```

```
    VAR dbox, bar, exit_cond, bar_word

    bar = READ_DATA_FILE@("/user/joe/my_menubar")
    SET_SELECTIONS@(45,bar)
    dbox = DB_LOAD@("db_bar")
    DB_MENU_BAR@(dbox,45)

again:
    DB_DISPLAY@(dbox)
    IF DB_CANCELLED@(dbox)
            RETURN
    exit_cond = "DB_EXIT_CTRL@(dbox)
    IF exit_cond = "menu_bar_"
    {
            bar_word = DB_MENU_BAR_WORD@(dbox)
            IF bar_word = "my_macro"
                    my_macro
            ELSE IF bar_word = "not_ok"
                    DB_CTRL_GRAYED@(dbox,"OK",TRUE)
            ELSE IF bar_word = "ok"
                    DB_CTRL_GRAYED@(dbox,"OK",FALSE)
    }
    GOTO again
ENDMACRO
```

Making Menu Options Grayed or Toggled

If you want, you can make menu options grayed or you can make them toggle options. These menu option attributes are set using the DB_MENU_STATUS@ macro:

```
    DB_MENU_STATUS@(dbox, name, value)
```

dbox is the name of the dialog box variable.

name is the name of the macro that is called by the menu option for which you want to specify an attribute. For instance, if the menu

option Delete calls the macro named "delete_it," then you specify delete_it as name to reference the Delete menu option.

value is a number that corresponds to the menu option attribute you want to set:

0 The menu option is displayed normally.

1 The menu option is grayed.

2 The menu option is toggled on and the toggle (a check mark) appears to the left of the menu option in the pull-down menu.

3 The menu option is made a toggle option, but the option is toggled off.

4 The menu option is made a toggle option, but the option is neither toggled on or off.

5 The menu option is a toggle option and a radio button is used as the toggle. The radio button appears to the left of the menu option in the pull-down menu.

6 The menu option is not displayed on the pull-down menu.

For example, the following macro makes a menu option grayed:

```
DB_MENU_STATUS@(dbox,"exit_doc",1)
```

Maintaining the Dialog Box Display

In some cases, you may find that you want a dialog box to remain displayed in situations when it would normally be exited. For example:

- Usually, push buttons of type Execute & Dismiss cause a dialog box to be exited when they are selected. You might find, however, that you would like to include Execute & Dismiss push buttons that perform an action without exiting the dialog box. For instance, you might have a dialog box in which you need to validate entries when a button of type Execute & Dismiss is selected. You only want to exit the entry box if all entries are valid. If an entry is invalid, you want the dialog box to remain displayed so that the user can type a valid entry.

- Certain conditions might involve exiting and redisplaying a dialog box without having first suspended dialog box operation. For instance, you may have an error handler that displays an error message and then sends processing back to a point in your dialog box macro before the DB_DISPLAY@ macro. In such instances, you need to maintain the dialog box display so that it can continue to be used.

You specify that a dialog box be maintained using the DB_WINDOW_REMAIN@ macro. The format for the macro is:

```
DB_WINDOW_REMAIN@(dbox, value)
```

dbox is the name of the dialog box variable.

value can be TRUE or FALSE. A value of TRUE indicates that the dialog box display is maintained when a push button of type Execute & Dismiss is selected or the dialog box is redisplayed using DB_DISPLAY@. A value of FALSE indicates that the dialog box is not maintained.

Ownerless Dialog Boxes

Most dialog boxes have a ‘‘parent’’ window. That is, the dialog box is called as a child task of another window or dialog box. For example, the Character Settings dialog box in Applix Words has the Words window as its parent. A parent window cannot be closed if any dialog boxes associated with it are active. This ensures that all dialog boxes (and their associated tasks) be completed before a parent window is exited.

Sometimes, you might want to create a dialog box that is not associated with a parent window. For example, if you created a stand-alone application using ELF, you might want the dialog box representing the application to be an ownerless dialog box. You define an ownerless dialog box using the DB_OWNERLESS@ macro. The format for the macro is:

```
DB_OWNERLESS@(dbox, value)
```

dbox is the name of the dialog box variable.

value can be TRUE or FALSE: TRUE indicates that the dialog box is ownerless, FALSE indicates that the dialog box has a parent window or dialog box.

Unless you specify otherwise using DB_OWNERLESS@, dialog boxes are assumed to have a parent window.

Dialog Box Set-aside Icons

You can specify the icon to be used when a dialog box is set aside. If you do not specify a set-aside icon for a dialog box, the standard set-aside icon for your window management software is used.

You create the graphic for a set-aside icon using the Bitmap Editor. When creating a bitmap to be used as a set-aside icon, you must specify the bitmap size. The size you make an icon should correspond to the icon size setting in the Customize Look and Feel dialog box. You can set the icon size to 32x32 pixels or 50x50 pixels. If you want to create an icon that will be displayed regardless of the icon size setting, create two separate icons corresponding to the two icon sizes.

When you name the bitmap file that is to be used as a set-aside icon, you must use the following naming convention:

- The first part of the bitmap file name must be a numerical name. For example, you could name the bitmap file "34."
- The second part of the bitmap file name must be one of the following strings that indicate the size of the bitmap:

  -32x32 The bitmap image is 32x32 pixels.

  -50x50 The bitmap image is 50x50 pixels.

The following are examples of valid file names for bitmaps that are to be used as set-aside icons:

```
45-50x50
1-32x32
```

The Bitmap Editor automatically appends the .im extension to a file name when the bitmap file is saved.

You use the DB_ICON@ macro to specify the set-aside icon that is to used for a dialog box. The format for this macro is:

```
DB_ICON@(dbox, bitmap_number)
```

dbox is the name of the dialog box variable.

bitmap_number is a number corresponding to the numerical file name you gave the bitmap file when you saved it. Do not use the string

portion of the file name (-50x50, for example) or the .im extension when you specify bitmap_number. For example, the bitmap image in the file 45-50x50.im would be specified as follows:

```
DB_ICON@(dbox,45)
```

When DB_ICON@ looks for the bitmap file, it automatically appends a string corresponding to the Icon size setting in the Customize Look and Feel dialog box to bitmap_number. For instance, if you specify 35 as bitmap_number and the Icon size setting is 16x16, DB_ICON@ looks for the bitmap file 35-16x16.im.

DB_ICON@ automatically looks in your macros directory for the bitmap file. If the file is not found in your macros directory, the axlocal directory is checked. If the file is not found there, the axdata/elf directory is checked. If the file is not contained in any of these directories, you must supply a full pathname as the argument to DB_ICON@ to indicate where the file is located.

The DB_ICON@ macro must be placed after the DB_LOAD@ macro, but before the DB_DISPLAY@ macro in your dialog box macro.

The following macro shows DB_ICON@ being used to specify a set-aside icon. The bitmap image for the icon is named "23."

```
MACRO DboxMacroDlg
    VAR dbox, color_vals
    dbox = DB_LOAD@("dbox_file")
    color_vals = "white", "red", "green", "yellow", "blue", "orange"
    DB_CTRL_STRINGS@(dbox, "Color", color_vals)
    DB_ICON@(dbox,23)
    DB_DISPLAY@(dbox)
    IF DB_CANCELLED@(dbox)
        RETURN
ENDMACRO
```

Calling a Dialog Box from Another Dialog Box

Usually, only one dialog box can be displayed at a time. Running a dialog box is an independent Applixware task that must be resolved before another task, such as displaying another dialog box, is run. However, if you want to display a new dialog box while the current dialog box is still running, you can use PEND_FOR_NEW_TASK@ to suspend operation of the current dialog box task and run a new task. When the new task is completed, processing resumes in the task containing the PEND_FOR_NEW_TASK@ macro.

The format for the PEND_FOR_NEW_TASK@ macro is:

```
PEND_FOR_NEW_TASK@(macro,args)
```

macro, a string, is the name of the macro that runs the task you want to perform.

args, a string or string array, is any argument or arguments to a macro.

Note that the task specified by PEND_FOR_NEW_TASK@ must be completed before processing returns to the calling task.

NOTE: When you use PEND_FOR_NEW_TASK@ to display two dialog boxes concurrently, make sure that each dialog box has a unique name (as specified when you create the dialog boxes).

If you want to exchange values between a dialog box and another task, such as another dialog box, you must use system variables for the values you want to exchange. You declare system variables using SET_SYSTEM_VAR@:

```
SET_SYSTEM_VAR@(name, value)
```

System variables retain their values across tasks. For example, if you want to use the value of a dialog box control in a different dialog box, you can assign the value to a system variable.

For example, suppose you have a push button in a dialog box that displays another dialog box when it is pressed. To display the new dialog box without exiting the current dialog box, you need to use PEND_FOR_NEW_TASK@.

The following example dialog box contains an OK button, a Cancel button, and a Display push button that displays a new dialog box. The new dialog box contains an entry box and an OK button. When the Display button is pressed, the new dialog box is displayed. After information is typed into the entry box in the new dialog box and the dialog box is exited, processing returns to the calling dialog box. The calling dialog box then displays an information box that shows the string that was typed in the other dialog box.

The macro that includes the PEND_FOR_NEW_TASK@ call is as follows:

```
MACRO CallDbox
        VAR dbox, exit_cond, other_val
        dbox = DB_LOAD@("dbox1_file")
        DB_CTRL_RETURN_ON_CHANGE@(dbox,"Display",
TRUE)
        DB_WINDOW_REMAIN@(dbox,TRUE)
 again:
        DB_DISPLAY@(dbox)
        IF DB_CANCELLED@(dbox)
                RETURN
        exit_cond = DB_EXIT_CTRL@(dbox)
        IF exit_cond = "Display"
        {
                PEND_FOR_NEW_TASK@("NewDbox")
                other_val = SYSTEM_VAR@("entry_val")
```

```
                INFO_MESSAGE@(
                        "Value from other dialog box: "
++other_val)
                GOTO again
                }
ENDMACRO
```

The "NewDbox" macro for displaying the other dialog box is as follows:

```
MACRO NewDbox
        VAR dbox2, entry_val, the_val
        dbox2= DB_LOAD@("dbox2_file")
        DB_DISPLAY@(dbox)
        IF DB_CANCELLED@(dbox2)
        {
                the_val = "Cancel was pressed"
                SET_SYSTEM_VAR@("entry_val", the_val)
                RETURN
        }
        the_val = DB_CTRL_GET_VALUE@(dbox2, "Entry")
        SET_SYSTEM_VAR@("entry_val", the_val)
ENDMACRO
```

Pokes

Pokes allow you to send a string from one ELF macro to another. For example, an application might have a dialog box that indicates when new mail has been received. When the application detects new mail, it uses a poke to send a message to the dialog box to indicate the arrival of the mail. The dialog box could handle this message by displaying an icon indicating that new mail has been received.

Pokes have two parts:

- A poke code is a number less than 10,000.
- A poke message is a string or array. This information can be displayed or used for interprocess communication.

A poke code is required. A poke message may have a NULL value. The following macro sends and receives pokes.


```
MACRO pokes

    VAR dbox, exit_cond, ctrl_value, done, code_array
    dbox = DB_LOAD@("pokes.d")

/*
*   The code array below is a set of Poke codes that are used by
*   DB_ACCEPT_POKES@.
```

```
*/
        code_array= 500, 501
        DB_CTRL_RETURN_ON_CHANGE@(dbox,"Button",TRUE)
/*
*       DB_ACCEPT_POKES@ establishes that this macro
*       receives pokes, and specifies which poke codes are received.
*       Any codes received that are not specified in the code_array parameter are
*       ignored.
*/
        DB_ACCEPT_POKES@(dbox, code_array)
        done = FALSE

/*
*       With DB_ACCEPT_POKES@ in effect, an exit condition is
*       generated each time a poke code matching an element in the code_array
*       is received.  In this example, an exit condition is generated when poke
*       codes 500 and 501 are received.
*/
        WHILE NOT done

                DB_DISPLAY@(dbox)
                IF DB_CANCELLED@(dbox) RETURN
                exit_cond = DB_EXIT_CTRL@(dbox)

                CASE of exit_cond

                        CASE "OK"
                                done = TRUE

/*
*       When the user presses the send button, the contents of the top edit
*       box is sent out as a poke.
*/
                CASE "Button"
                        DB_SEND_POKE@(500,
DB_EDITBOX_GET_DATA@(dbox, "Editbox"))

/*
*       The poke_ exit condition, which is triggered when the specified poke codes
```

```
*       (500 and 501) is handled here. The poke message data is written to an edit
*       box.
*/
                        CASE "poke_"
                        if DB_GET_POKE@(dbox) = 501
                                DB_EDITBOX_SET_DATA@(dbox, "Editbox.1",
                                DB_GET_POKE_DATA@(dbox))
                ENDCASE
        WEND
ENDMACRO
```

Poke Macros

The following macros are useful for sending and receiving pokes:

- DB_SEND_POKE@(poke_code, message) sends a message. poke_code is a number below 10000. message is either a string or an ELF array. The string may be NULL. Often, the macro receiving the poke is only concerned with the poke code. message is not always required.

- DB_ACCEPT_POKES@(dbox, poke_codes) configures a dialog box to accept one or more poke codes. dbox is the dialog box handle returned from DB_LOAD@. poke_codes is an array of codes that the dialog box macro should accept.

  For example, the following indicates that a dialog box recognizes a single poke code:

  ```
  codes[0] = 5
  DB_ACCEPT_POKES@(dbox,codes)
  ```

  The following indicates that a dialog box recognizes four poke codes:

  ```
  codes = 5, 10, 15, 33
  ```

DB_ACCEPT_POKES@(dbox,codes)

- When a message is received, the exit condition poke_ is automatically triggered. (The exit condition is all lower-case with a trailing underscore.) You include statements in your receiving dialog box macro to test whether the poke_ exit condition is present.

- If your macro accepts more than one poke message, DB_GET_POKE@ returns the poke code. You can use conditional statements to delineate between one poke code and another. An example follows:

```
exit_cond = DB_EXIT_CTRL@(dbox)
IF exit_cond = "poke_"
{
        poke = DB_GET_POKE@(dbox)
        IF poke = 10
        {
                count = count + 1
                DB_CTRL_VALUE@(dbox, "Count", count)
                GOTO again
        }

        IF poke = 20
        {
                DB_CTRL_GRAYED@(dbox, "Count", TRUE)
                GOTO again
        }
}
```

- DB_GET_POKE_DATA@ returns the poke message string. You can use IF-THEN statements or CASE statements in your dialog box macro to perform operations based on the message received. The poke message string is a maximum of 950 bytes in length.

Creating and Changing Dialog Boxes Using Macros

Table 10-3 lists dialog box macros that you can use to manipulate dialog box features. These features are normally manipulated using the dialog box editor.

You can use these macros to create a dialog box dynamically. This means that you do not have to store your dialog box in a file. For a description on how to use any of these macros, refer to the online help for each of the macros.

Table 10-3. Macros that Create and Change Dialog Boxes

Macro	Description
DB_CREATE_DIALOG@	Creates a dialog box having a specified title, width, and height.
DB_TITLE@	Sets the title to be used for the dialog box.
DB_WIDTH@	Sets the width, in pixels, of the dialog box.
DB_HEIGHT@	Sets the height, in pixels, of the dialog box.
DB_CREATE_CTRL@	Creates a control.
DB_CTRL_BUTTON_TYPE@	Indicates the type of a push button. The type can be Normal, Execute & Dismiss, Dismiss, Execute, or Bitmap.
DB_CTRL_HEIGHT@	Sets the height for list boxes (number of lines).
DB_CTRL_LENGTH@	Sets the maximum number of characters that can be displayed in a list box.
DB_CTRL_LINE_THICKNESS@	Indicates the thickness with which a panel/line control is drawn.
DB_CTRL_MULTI_SELECT@	Indicates whether multiple selections are allowable in a list box.

Table 10-3. Macros that Create and Change Dialog Boxes (cont.)

Macro	Description
DB_CTRL_NO_TITLE@	Indicates whether the title of an entry box should be displayed as a label next to the entry box.
DB_CTRL_OPTIONAL@	Indicates whether an entry box entry is optional or mandatory.
DB_CTRL_PICK_DEFAULT@	Indicates whether double-clicking on a list box item will choose the item and the default push button.
DB_CTRL_TITLE@	Sets the title for a control.
DB_CTRL_TRIM@	Indicates whether leading or trailing spaces should be trimmed from entry box strings.
DB_CTRL_WIDTH@	Sets the display width for entry boxes, push buttons, or list boxes.
DB_CTRL_XPOS@	Sets the position, with reference to the x-axis, of the top left corner of a control.
DB_CTRL_YPOS@	Sets the position, with reference to the y-axis, of the top left corner of a control.

11 The Bitmap Editor

This chapter covers the following topics:

- Creating a new bitmap image
- Changing an existing bitmap image
- Using the Bitmap Editor

Starting the Bitmap Editor

The Bitmap Editor is used to create and change bitmap images. You can use bitmap images as the graphic element in a bitmap push button or as a decoration in a dialog box.

In addition to creating bitmaps using the Bitmap Editor, you can create an image in Applix Graphics and create a bitmap image from the Graphics file. To start the Bitmap Editor and create a new bitmap file:

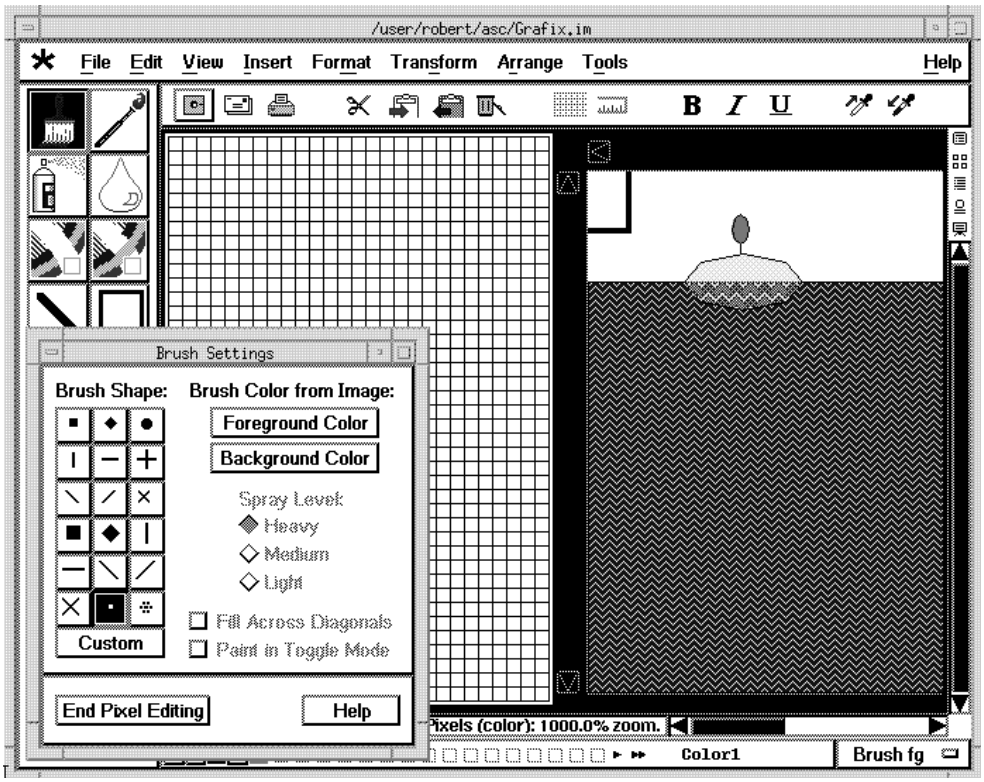
1. Open or move to a Macro Editor window.
2. Choose **Tools → Create Bitmap**. The Create Bitmap dialog box is displayed.
3. Type a name for the bitmap image in the File name entry box.
4. In the In directory entry box, type the path for the directory where you want the bitmap image file placed. The name of your macros directory is displayed in the *In directory* entry box when you display the dialog box. If you want to place the bitmap image file in your macros directory, do not change the pathname in the entry box.
5. In the Image width entry box, type the width, in pixels, that you want the bitmap image to be.
6. In the Image height entry box, type the height, in pixels, that you want the bitmap image to be.
7. Turn on Color if the Bitmap is a color image.
8. Click on **OK** to exit the dialog box and display the Bitmap Editor.

To edit an existing bitmap image file, choose **Tools → Change Bitmap** in the Macro Editor. When you choose this option, the Open dialog

box is displayed and lists all the bitmap image files in your macros directory. Double-click on the bitmap image file you want to edit.

Using the Bitmap Editor

The following is the Applix Bitmap Editor:



The Brush Settings dialog box gives your choice of brush shape indicates how many pixels are affected with each application of the brush in the edit area.

The *edit area* displays the individual pixels for the bitmap image. The number of pixels shown depends on the size of the image specified when you created the bitmap.

To draw a bitmap image:

1. Select a brush shape from the brush area.
2. Click on a pixel in the edit area to reverse its color. For example, clicking on a white (background) pixel changes it to a black (foreground) pixel. The number of pixels that are affected by each mouse click depends on the brush shape you have chosen.
3. You can drag the brush in the image area to apply the brush to many pixels.
4. To exit the Bitmap Editor and save any changes you have made to the bitmap image, choose **File → Exit**.

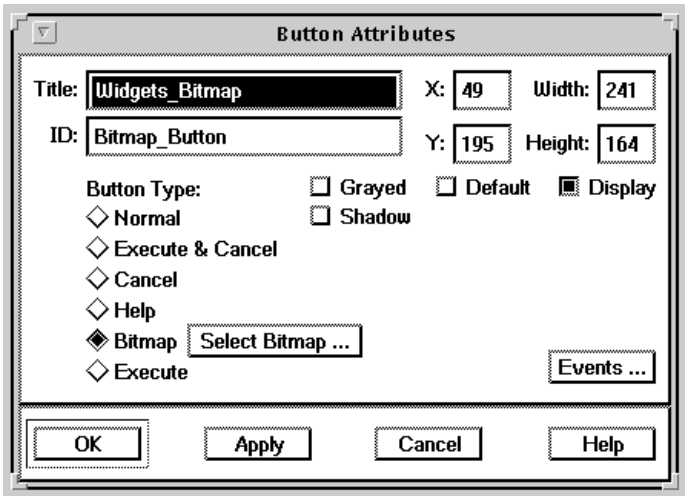
   To exit the Bitmap Editor without saving any changes you have made to the bitmap image, choose **File → Discard.**

For information about including bitmaps in push buttons or in dialog boxes, see the section ‘‘Adding Controls.’’

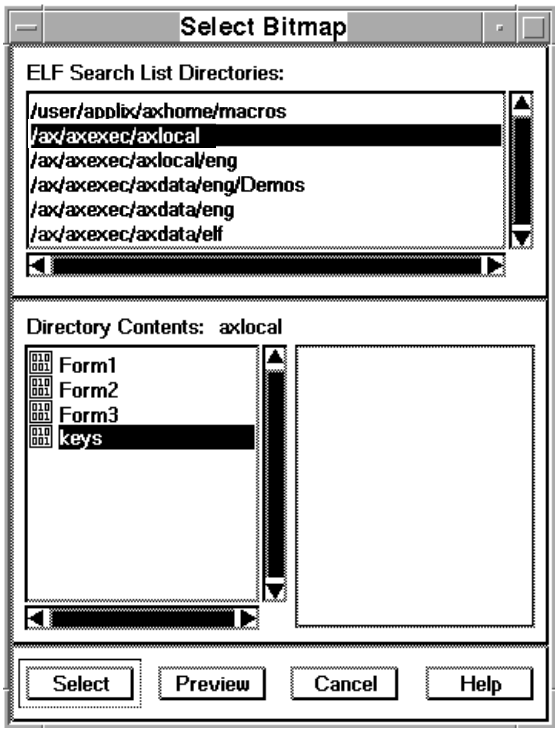
Assigning a Bitmap to the Bitmap Button

To assign a bitmap to the bitmap button, follow these steps:

1. From the Button Attributes dialog box, choose Bitmap as the Button Type. The Select Bitmap option appears:



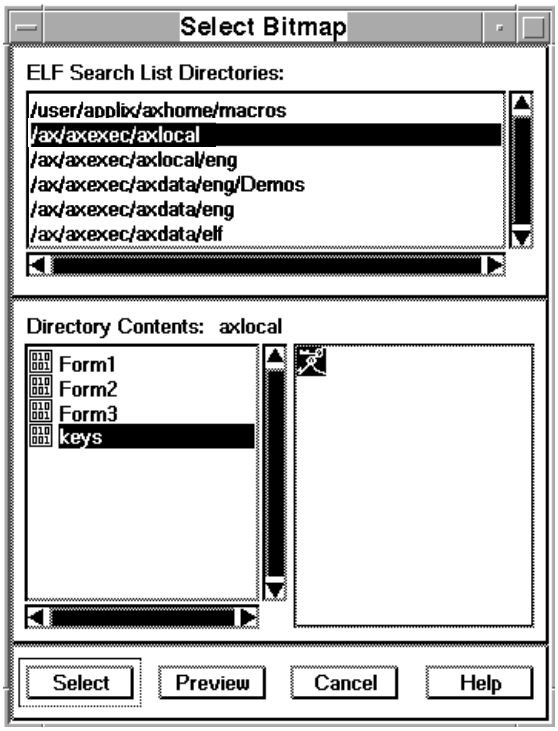
2. Click on the Select Bitmap option. The Select Bitmap dialog box appears.



The Dialog Box Editor displays a list of possible directories containing bitmaps, beginning with your macros directory. If the bitmap you want is not in these directories, ELF will not be able to find it.

Use the Preview button in the Select Bitmap dialog box to preview a bitmap image before you place it in the button. To preview a bitmap image:

1. Click on a bitmap name in the Directory Contents list area.
2. Click on Preview.



The bitmap image is displayed to the right of the Directory Contents list area.

The Dialog Box Editor uses the ELF search path determined by the login.am file located in your macros directory. The login.am file contains the elf_search_list@ system variable which contains a list of path names that replace the default ELF search path. You can add or delete directories from the search path to widen or narrow your search. ELF looks at the directories named in this variable for macros, image files, dialog boxes, and object files. For example, suppose your login.am file contained the following lines:

```
names =     "/user/applix/axhome/macros",
            "/ax/lpgms",
```

```
            "/ax/pgms/axelf"
    set_system_var@("elf_search_list@",names)
```

The ELF functions look for information only in the three named directories. Note that this example overwrites the previously-established elf_search_list@ system variable.

The Dialog Box Editor automatically chooses the first bitmap in the Directory Contents list box. You can scroll through the list box to choose the desired bitmap. After you choose a bitmap:

1. Click **Select** in the Select Bitmap dialog box.

2. Click **OK** in the Button Attributes dialog box.

The bitmap image appears in the button.

You can create a bitmap by choosing Tools → Create Bitmap from the Macro Editor. You can also export documents from Applix Graphics in Applix Bitmap format.

12 ELF Tasking

Applixware is a single-threaded, multi-tasking application programming environment. This chapter discusses the ELF tasking system, and the features of the ELF tasking system that you can manipulate through ELF macros.

- Applixware tasking system
- ELF scheduler
- ELF tasking macros

Applixware Tasking System

Applixware is a single UNIX process which contains a complete, non-preemptive multi-threaded programming environment. The Applixware tasking system allows you to schedule and run up to 100 lightweight threads from within the Applixware UNIX process.

In Applix terminology, execution threads are called *tasks*.

Each Applixware application, such as Words, Graphics, and so on, consists of multiple tasks. These application are carefully designed to promote efficiency and responsiveness through the use of multiple tasks. Multiple tasks are also used within Applixware to allow *non--modal* dialog boxes, which are dialog boxes which stay active and visible for long periods of time.

Since Applixware is a non-preemptive multitasking environment, before developing an application, you should ask yourself:

1. Will the program I am writing require a long time to run?
2. Is the operation I am performing one which must be performed synchronously with respect to some other operation, or can it be performed asynchronously or in parallel?

Most ELF macros execute quickly, and can be run synchronously. These kinds of programs execute as subroutines of the Applix task from which they are called.

For example, suppose you write a new spreadsheet function called sumA1andB20(). The function adds the value of cells A1 and B20, and returns the sum. After this function is written and tested, it can be used within a spreadsheet cell as part of a spreadsheet expression. When the spreadsheet is calculated, the ELF function runs as a spreadsheet subroutine within the spreadsheet task. Since the function is *safe* (see the definition of a safe macro described later in this document)

and fast, running synchronously as a subroutine within the spreadsheet task is the correct approach.

You can also write complex programs which post dialog boxes, perform I/O, do other lengthy calculations or, in the case of Applix Spreadsheets, call *unsafe* macros. Unsafe macros are described later in the *ELF Spreadsheets Macros* guide. In these cases, you may need to break the program into multiple tasks, and surrender the thread of control at frequent intervals so that other Applixware tasks can run.

For example, suppose you write a spreadsheet function that:

1. Tests to see if the number of shares of a particular stock exceeds 100,000
2. Queries a database for the trading volume of the stock over the last 120 days
3. Inserts the data into another section of the spreadsheet.

Such a macro is "unsafe". It is attempting to edit the spreadsheet as it is being calculated, and it is lengthy. You cannot predict how long the database query will take.

In this case, you should break the program into two tasks. The first part, which tests to see whether the volume exceeds 100,000 shares, can be safely executed in the current spreadsheet task. The second part should be called using the ELF tasking macro NEW_TASK_UNPENDED@. This guarantees that spreadsheet tasks are not prevented from completing their calculation loop by a lengthy calculation such as a database query, and that the spreadsheet is not edited while it is calculating.

Starting New Tasks

Any program or macro which is attached to a button or menu pull-down starts a new task. Any macro started by pressing F8 and entering the names of the macro, starts a new task.

A user-written macro inserted into a spreadsheet cell does *not* start a new task. Such macros run as subroutines of the spreadsheet.

There are three ELF macros that create new tasks. These macros differ from each other in the way that the child task controls the parent task's thread of execution. These macros are provided to give the programmer methods for managing the flow of control within Applixware. The following ELF macros start new tasks:

- NEW_TASK@ creates a new task, and the parent task (the task that called NEW_TASK@) is suspended. The parent is blocked on the child task until the child task completes. However, the parent task is unblocked if the child task gives up its thread of control. Unblocking the parent task implies that the parent task resumes execution some time in the future *before* the child task runs again.

  The child can explicitly give up control of the execution thread by calling DELAY@. (There are other ELF macros that also give up control of the execution thread. These are described later in this chapter.) When the child task gives up control of the thread, the Applix scheduler runs the next task in its queue. Note that this is not necessarily the parent task. Eventually, the parent will run again.

  NEW_TASK@ returns zero (0) if the child task terminates. If the child task gives up control of the execution thread, NEW_TASK@ returns the Task ID of the child task.

- PEND_FOR_NEW_TASK@ creates a new task, and the parent task (the task that called PEND_FOR_NEW_TASK@) is suspended.

The parent task is blocked until the child task exits. The parent task is not unblocked even if the child gives up its thread of control. If the child gives up control, other tasks can run, but the parent remains blocked.

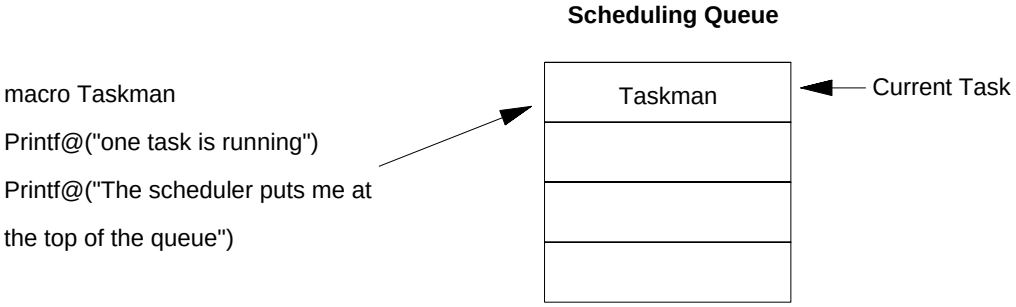
PEND_FOR_NEW_TASK@ returns the data returned by the child task.

- NEW_TASK_UNPENDED@ creates a new task, but it does not surrender control of the execution thread. The parent continues to run. The child task is added to the Applix scheduler queue. The child does not run until some later time when the parent surrenders the thread.

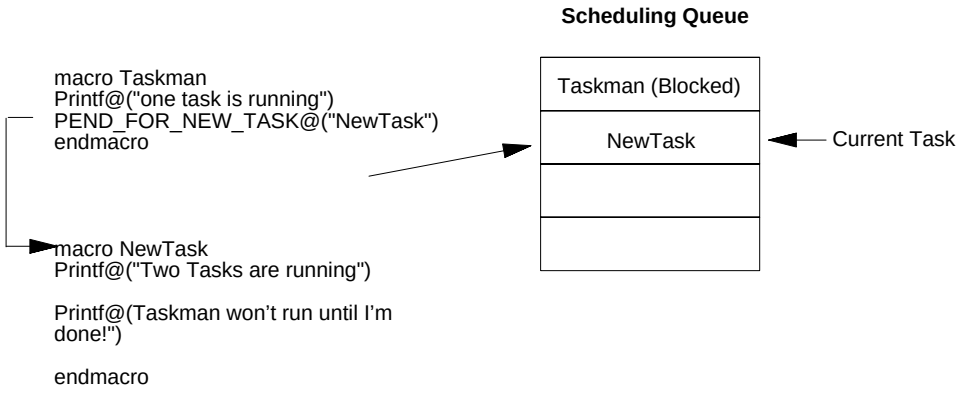
Applixware Scheduler

You can run more than one ELF task at a time. The macros NEW_TASK@, PEND_FOR_NEW_TASK@, and NEW_TASK_UNPENDED@ start new ELF tasks. When one of these macros runs, it adds a task to the ELF scheduling queue.

The ELF scheduling queue is a list of currently-running ELF tasks that is maintained by the ELF scheduler. When a task is created, it is added to the queue. The following figure illustrates this.

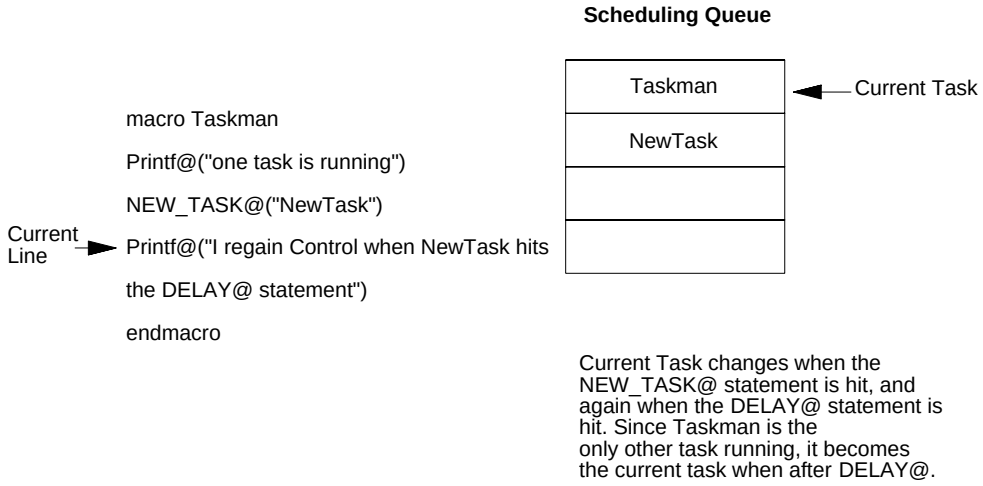


If a running task spawns a new task, the new task is added to the queue. In the following figure, the PEND_FOR_NEW_TASK@ macro suspends the execution of Taskman until its child task is completed.



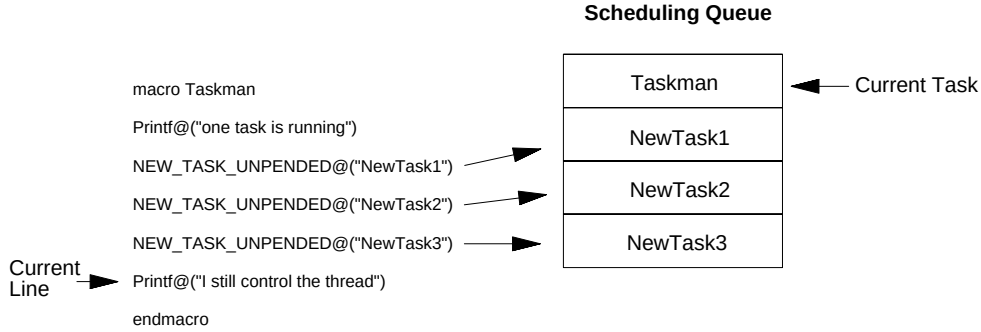
NEWTASK@

If you spawn a new task with NEW_TASK@ , the child task can give up control of the execution thread by calling either DELAY@, or DB_DISPLAY@. The following figure shows this transaction.



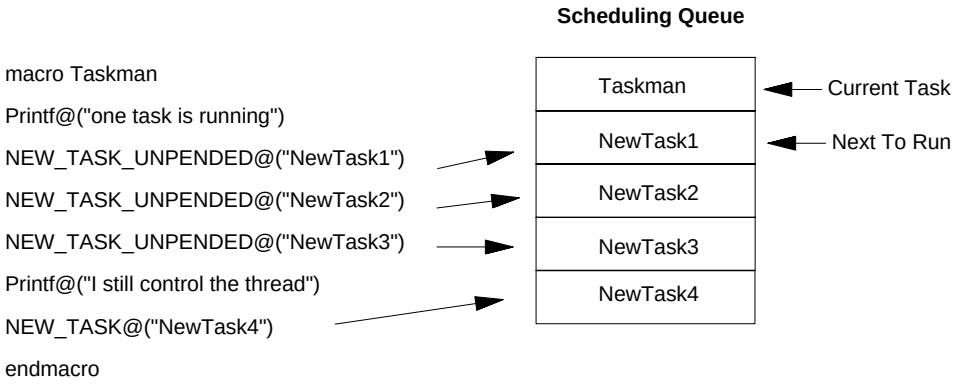
NEW_TASK_UNPENDED@

Both NEW_TASK@ and PEND_FOR_NEW_TASK@ pass control of the thread to a child task when they are called. You can add tasks to the scheduling queue, but maintain control of the thread with NEW_TASK_UNPENDED@. The following figure shows a macro that adds three new tasks to the scheduling queue, but maintains control of the thread.



The figure shows that three tasks have been added to the scheduling queue, but the current task has not changed. When the Taskman macro exits, runs DELAY@ or runs DB_DISPLAY@, the scheduler changes the current task. Note that macros added to the scheduling queue using NEW_TASK_UNPENDED@ are run by the scheduler in the order that they are queued.

If a fourth task were added to this list with NEW_TASK@ , it would be added to the bottom of the queue. The following figure illustrates this:



When NEW_TASK@ runs, Taskman is suspended, and the ELF Scheduler starts to run the other tasks in its queue. The first task that the ELF Scheduler runs is NEWTASK1, which was loaded onto the queue with NEW_TASK_UNPENDED@ .

ELF Tasking Macros

Table 12-1 shows macros that are important in spawning and controlling ELF tasks.

Table 12-1 ELF Tasking Macros

MACRO	DESCRIPTION
TIME_SLICE@	Passes control to the ELF scheduler after a specified period of time. This macro is used in conjunction with the pragmas @@@TIME_SLICE and @@@NO_TIME_SLICE.
DELAY@	Passes control of the execution thread to the ELF scheduler.
NEW_TASK@	Starts a new task.
PEND_FOR_NEW_TASK@	Starts a new task, and waits for its completion.
NEW_TASK_UNPENDED@	Schedules a new task without giving up control of the execution thread.

Macros that Yield Control of the Thread

Whenever one of the following macros is encountered in your ELF macro, your macro gives up control of the execution thread.

- DELAY@
- PROMPT@
- INFO_MESSAGE@
- WORK_IN_PROGRESS@
- YES_NO_PROMPT@
- YES_NO_HELP_PROMPT@
- YES_NO_CANCEL_PROMPT@
- OPEN_PROMPT@
- TIME_SLICE@

Parent Tasks

There are two different kind of parent tasks: ELF parent tasks and macro parent tasks. Every ELF Task has an ELF parent task. Most ELF tasks have a macro parent task. This section describes the differences between them.

ELF Parent Task

An ELF parent task is the task that spawns another task. For example, if you run an ELF macro called One, and it spawns a new task called Two by calling NEW_TASK@, One is the ELF parent task of Two.

An ELF task has the same ELF parent task throughout its life. You retrieve the ID of the ELF parent task through the macro ELF_PARENT_TASK@.

NOTE: The ELF parent task has no relationship with its child tasks aside from spawning them, so knowing a task's ELF parent is not usually significant. Programmers usually find that the macro parent of a task is a much more significant piece of information than the ELF parent.

Macro Parent Task

If a macro parent task is an Applixware application, the macro parent affects its child tasks in three ways:

- Child tasks terminate if the macro parent task terminates.
- Child tasks are iconified when the macro parent task is iconified. The child inherits its set-aside icon from the macro parent.
- Child tasks use the macro parent tasks as the default window for output.

Unlike ELF parent tasks, the macro parent task can be changed. Run the macro SET_MACRO_PARENT_TASK@ to change the macro parent task. The following section shows an example of this.

Example: Changing the Macro Parent Task

Changing the macro parent task in an ELF macro allows you to use two Applix documents for output. The following procedure illustrates this:

1. Open two Applix Words windows.
2. Open the Macro Editor and enter the following macro:

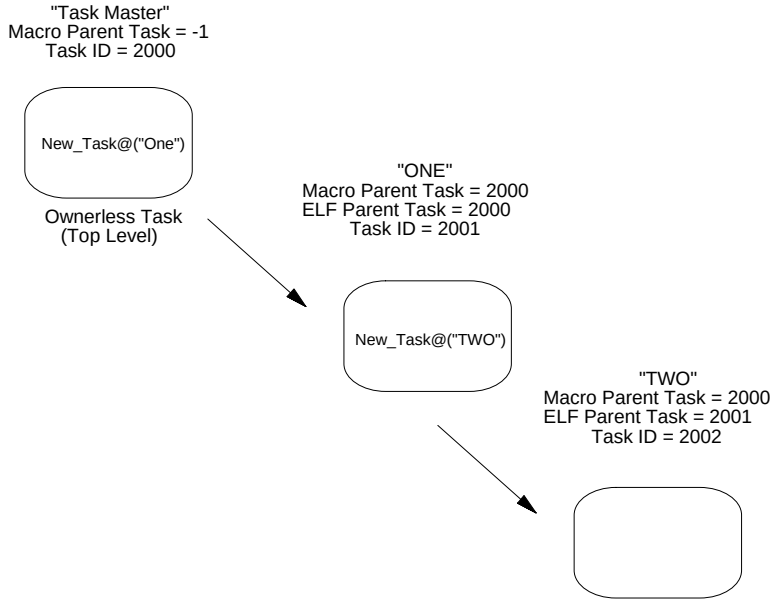
```
macro check_words_task_id
info_message@("The words task id is "++Macro_Parent_Task@())
endmacro
```

3. Compile this macro.
4. Exit the Macro Editor.
5. Click in one of the words windows, then press **F8** and run **check_words_task_ID**. Write down the id.
6. Click in the other Words window, and run **CHECK_WORDS_TASK_ID@**. Write down the id. You should have the task IDs of both words tasks written down.
7. Open the Macro Editor. Type in the following macro:

```
macro Say_Hello

set_macro_Parent_task@(1658)     ' the number in parentheses is the
task ID of the
                                                    ' Words window.

WP_TYPE@("Hello")
endmacro
```

8. You can use this macro to write the string Hello to either of the Words windows by entering the task ID of the target window as an argument to SET_MACRO_PARENT_TASK@, then re--compiling.

When you create a chain of ELF tasks, the macro parent tasks are inherited from the parents. The following illustrates this:



Note that the macro parent task of Two is Task Master, not One. The default macro parent task for an ELF task is always the top task in its chain, whether that task is an application, like a spreadsheet, or another ELF macro.

Parentless Tasks

An ELF task can declare itself to be parentless by running SET_MACRO_PARENT_TASK@ with an argument of 0.

```
MACRO Tasks_demo

VAR dbox, exit_cond, ctrl_value
Set_Macro_Parent_Task@(0)  'Make this guy the Task Master
```

```
info_message@("My Task ID is "++ELF_TASK_ID@()++
                        "\nMy Parent ID is "++MACRO_PARENT_TASK@())

ENDMACRO
```

Whenever an Applixware application is run, it is immediately made a parentless task, subservient to no other task.

Windowless Tasks

If you write an ELF macro that does not have an associated window, it is very important that your macro terminates cleanly. It is possible to write eternal macros which run even if you exit Applixware. For example, if you set up a windowless macro as a top-level task, it runs until it completes. If that macro contains an infinite loop, it runs forever. In this case, you must kill the task from UNIX.

Tasking Macros

This section describes all of the macros that return information about tasks, their IDs, and information about their parents

ELF_PARENT_TASK@

ELF_PARENT_TASK@ returns the ID of the task that spawned the current task. The information returned by ELF_PARENT_TASK@ is always the same throughout the life of the task.

Running ELF_PARENT_TASK@ from an application returns the ID of the intermediate task that starts the application. For example, suppose you run ELF_PARENT_TASK@ from a spreadsheet cell. ELF_PARENT_TASK@ returns the ID of the task that spawned the spreadsheet. This task is created by the macro SS_APPLICATION_DLG@(), and only exists to start the Spreadsheet application. After the spreadsheet is started, its parent task terminates. It is not visible in the Applixware task list.

NOTE: All applications are started by temporary tasks, which then terminate. Because of this, the information returned from ELF_PARENT_TASK@ is not very useful when this macro is run directly from an application.

ELF_TASK_ID@

ELF_TASK_ID@ returns the task ID of the current task.

KILL_TASK@

KILL_TASK@ deletes a task.

MACRO_PARENT_TASK@

MACRO_PARENT_TASK@ returns the parent task of the current macro. The parent task can be changed using macros such as SET_MACRO_PARENT_TASK@ and SELECT_WINDOW@.

SET_MACRO_PARENT_TASK@

SET_MACRO_PARENT_TASK@ establishes a new parent for the current task. If you call SET_MACRO_PARENT_TASK@ with an argument of 0, the task becomes parentless. This task continues to run, even if all other applications terminate.

TASK_LIST@

TASK_LIST@ returns an array containing the name and ID of all current Applixware tasks.

TASK_TO_WINDOW@

TASK_TO_WINDOW@ returns an active task's window ID.

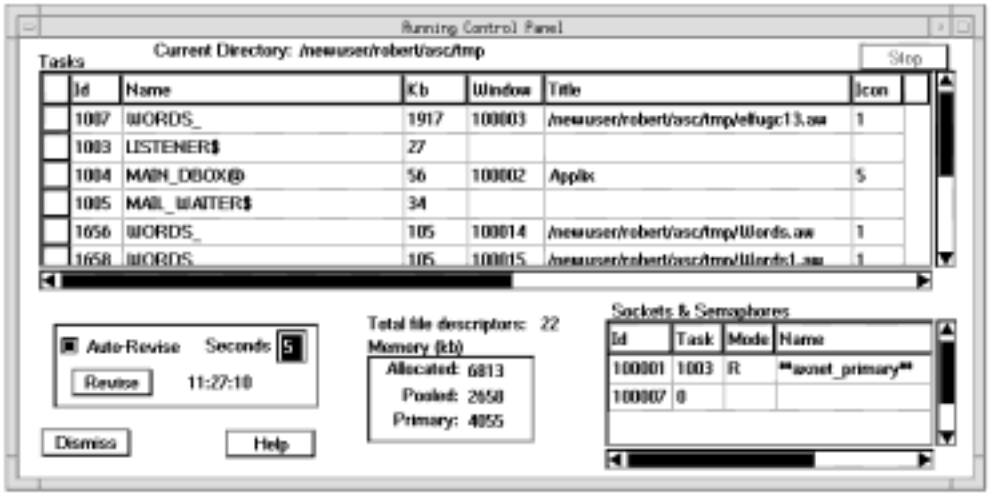
Tasks.am Demo Application

There is an application called Tasks.am in the Applixware install directory under Demos. This ELF macro displays a dialog box that lists all the currently running Applixware tasks. To run Tasks, follow these steps:

1. Press **F8**.
2. Enter **Tasks** in the entry area.

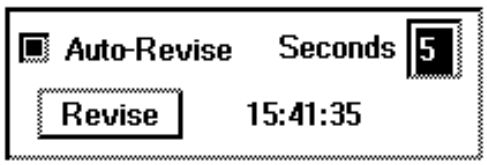
3. Press **OK**.

The following is an example of the Tasks dialog box:



Updating the Display

The box in the lower left-hand corner of the Tasks screen allows you to manually update the display to reflect new system information, or set an interval for updating the display.



To update the Tasks display manually, click Revise.

To set an interval, turn on the Auto-Revise toggle button, and enter the number of seconds.

Task Information

The main box in this dialog box shown the list of currently running Applixware tasks. The main dialog box contains the following fields:

- ID is the Task ID.
- Name is the name of the application (such as WORDS_) or the name of the ELF macro running as a task.
- KB is the number of kilobytes of memory occupied by the task.
- Window is the window ID of the task. By default, Applixware applications always have a window ID. However, you can open Applixware applications as windowless tasks. If a task is windowless, no window ID is displayed.
- Title is the title displayed at the top of the task window.
- Icon is the number of the set-aside icon assigned to the task.

Memory and File Descriptors

There are three pieces of information on memory allocation on the Tasks screen:

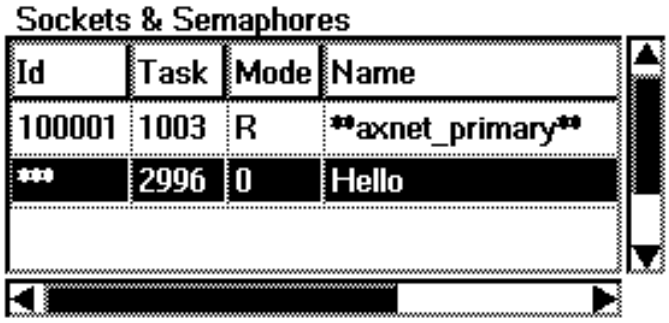
- *Allocated* is the total amount of memory used by all the Applixware tasks.
- *Pooled* is memory allocated by Applixware Task 0 for use by other tasks. Some of this memory may be in use by active tasks, and some may be free.
- *Primary* is the amount of memory used by Task 0, which is the main Applixware task. Task 0 keeps a list of currently active

tasks, allocates resources to those tasks, and arbitrates between tasks that are competing for resources.

Total File Descriptors is the number of UNIX file handles in use by Applixware.

Sockets and Semaphores

The sockets and semaphores table lists sockets and semaphore names that are currently in use by Applixware tasks.



In this example, the top entry is a socket, and the bottom entry is a semaphore. Sockets have the following information:

- An ID in the 100000 range
- An associated task ID
- A mode which is either Read (R) or Write (W)
- A name associated with the socket.

Semaphores have an associated task, and a name. Semaphores are added to the list through the ELF macro GET_RESOURCE@, and removed from the list through the ELF macro RELEASE_RESOURCE@.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W