TriTeal Enterprise Desktop (TED™) 4.0

Desktop Kornshell User's Guide

Printed in USA

Part No. TED004004000

Printing History
August 1995 ... First Edition
Printed in the U.S.A

Sales Information and Technical Support

TriTeal Corporation
2011 Palomar Airport Road, Suite 200
Carlsbad, CA 92009
USA

Tel: (619) 930-2077
Fax: (619) 930-2081
Support E-Mail: support@triteal.com
Sales E-Mail: info@triteal.com
World Wide Web: http://www.triteal.com/

Contents

Preface

The *Desktop KornShell User's Guide* provides the information you need to create Motif applications with KornShell (kshell) scripts. In addition to the basic information you'll need to get started, several example scripts of increasing complexity are described. Throughout this guide the term *dtksh* means the Desktop KornShell.

Who Should Use This Guide

This guide is intended for programmers who want a quick and easy means of creating Motif applications, but don't have the time, knowledge, or inclination to use the C programming language. A good understanding of kshell programming, Motif, the Xt Intrinsics, and, to a lesser extent, Xlib is needed. An understanding of C would also be helpful.

How This Guide Is Organized

Chapter 1, "Introduction to Desktop KornShell," describes the basic information you need to begin writing Motif applications in `dtksh` scripts.

Chapter 2, "A Sample Script," describes two simple `dtksh` scripts. The first script creates a push button widget within a bulletin board widget. The second script expands the first by adding a callback for the push button.

Chapter 3, "Advanced Topics," describes more advanced topics pertaining to `dtksh` scripts.

Chapter 4, "A Complex Script," describes a much more complex script than either of the ones described in Chapter 2. This script creates a graphic interface to the `find` command.

Appendix A, "dtksh Commands," lists all the `dtksh` commands.

Appendix B, "dtksh Convenience Functions," contains man pages for commands or functions that are not documented elsewhere.

Appendix C, "Listing for script_find," contains the complete listing of the complex script described in Chapter 4.

Related Books

The following books provide information on kshell programming, Motif, the Xt Intrinsics, and Xlib:

- *Desktop KornShell Graphical Programming For the Common Desktop Environment Version 1.0*, by J. Stephen Pendergrast, Jr., published by Addison-Wesley, Reading, MA 01867.
- *The New KornShell Command and Programming Language*, by Morris I. Bolsky and David G. Korn, published by Prentice-Hall, Englewood Cliffs, NJ 07632.
- *KornShell Programming Tutorial*, by Barry Rosenberg, published by Addison-Wesley, Reading, MA 01867.
- *OSF/Motif Programmer's Guide*, Open Software Foundation, 11 Cambridge Center, Cambridge, MA 02142, published by Prentice-Hall, Englewood Cliffs, NJ 07632.
- *OSF/Motif Programmer's Reference*, Open Software Foundation, 11 Cambridge Center, Cambridge, MA 02142, published by Prentice-Hall, Englewood Cliffs, NJ 07632.
- *OSF/Motif Reference Guide*, by Douglas A. Young, published by Prentice-Hall, Englewood Cliffs, NJ 07632.
- *Mastering OSF/Motif Widgets* (Second Edition), by Donald L. McMinds, published by Addison-Wesley, Reading, MA 01867.
- *The X Window System Programming and Applications with Xt OSF/Motif Edition*, by Douglas A. Young, published by Prentice-Hall, Englewood Cliffs, NJ 07632.

- *The Definitive Guides to the X Window System, Volume 1: Xlib Programming Manual*, by Adrian Nye, published by O'Reilly and Associates, Sebastopol, CA 95472.
- *The Definitive Guides to the X Window System, Volume 2: Xlib Reference Manual*, edited by Adrian Nye, published by O'Reilly and Associates, Sebastopol, CA 95472.
- *The Definitive Guides to the X Window System, Volume 3: X Window System User's Guide*, by Valerie Quercia and Tim O'Reilly, published by O'Reilly and Associates, Sebastopol, CA 95472.
- *The Definitive Guides to the X Window System, Volume 4: X Toolkit Intrinsics Programming Manual*, by Adrian Nye and Tim O'Reilly, published by O'Reilly and Associates, Sebastopol, CA 95472.
- *The Definitive Guides to the X Window System, Volume 5: X Toolkit Intrinsics Reference Manual*, edited by Tim O'Reilly, published by O'Reilly and Associates, Sebastopol, CA 95472.
- *The Definitive Guides to the X Window System, Volume 6: Motif Programming Manual*, by Dan Heller, published by O'Reilly and Associates, Sebastopol, CA 95472.

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
Code samples may display the following:		
`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`

Introduction to Desktop KornShell 1

Desktop KornShell (`dtksh`) provides kshell scripts with the means for easily accessing most of the existing Xt and Motif™ functions. `dtksh` is based on `ksh-93`, which provides a powerful set of tools and commands for the shell programmer, and which supports the standard set of kshell programming commands.

`dtksh` supports all the features and commands provided by `ksh-93`. In addition, `dtksh` supports a large selection of the `libDt` functions, most of the widget-related Motif functions, a large subset of the Xt Intrinsics functions, and a small subset of the Xlib functions. All the supported functions are listed in Appendix A.

Using Desktop KornShell to Create Motif Applications

This section describes how to use `dtksh` to create Motif applications. To successfully use `dtksh`, you should have experience with Xlib, the Xt Intrinsics, the Motif widgets, and KornShell programming. It is also helpful to know the C programming language. If you are not familiar with any of these, you should refer to the appropriate documentation. Even if you are familiar with these systems, you should have access to the applicable man pages for reference.

In addition, your system should have these libraries:

- `libDtHelp`
- `libDtSvc`
- `libX11`

- `libXm`
- `libXt`
- `libtt`

Resources

Resources are widget variables that you use to define attributes such as size, location, or color. Each widget usually has a combination of its own resources, plus resources it inherits from higher level widgets. Xt Intrinsics and Motif resource names consist of a prefix (`XtN` or `XmN`) followed by the base name. The first letter of the base name is *always* lowercase, and the first letter of subsequent words within the base name is *always* uppercase. The convention for resource names in `dtksh` scripts is to delete the prefix and use the base name. Thus, the resource `XmNtopShadowColor` becomes `topShadowColor`.

Some Xt and Motif commands allow the shell script to pass in a variable number of parameters, representing resource-value pairs. This is similar to the argument list passed to the corresponding Xt or Motif C function. Examples include any of the commands used to create a widget, plus the `XtSetValues` command. In `dtksh`, resources are specified by a string with the following syntax:

`resource:`*`value`*

where *resource* is the name of the resource and *value* is the value assigned to the resource. `dtksh` automatically converts the *value* string to an appropriate internal representation. For example:

```
XtSetValues $WIDGET height:100 width:200 resizePolicy:RESIZE_ANY
XmCreateLabel LABEL $PARENT myLabel labelString:"Close Dialog"
```

When you retrieve widget resource values using `XtGetValues`, the return value is placed in an environment variable. Thus, unlike the Xt Intrinsics, the `dtksh` version of `XtGetValues` uses a name: (environment) variable pair, rather than a name:value pair. For example:

```
XtGetValues $WIDGET height:HEIGHT resizePolicy:POLICY
      sensitive:SENSITIVE
echo $HEIGHT
echo $POLICY
echo $SENSITIVE
```

The preceding `dtksh` segment might produce this output:

```
100
RESIZE ANY
TRUE
```

Certain types of resource values, including string tables and bit masks, have special representation. For example, the List widget allows a string table to be specified for both the `items` and `selectedItems` resources. In `dtksh`, a string table is represented as a comma-separated list of strings, which is similar to how Motif treats them. When a resource that returns a string table is queried using `XtGetValues`, the resulting value is a comma-separated set of strings.

A resource that expects a bit mask value to be passed to it expects the mask to be specified as a string composed of the various mask values separated by the |(bar) character. When a resource that returns a bit mask is queried, the return value is a string representing the enabled bits, separated by the | character. For example, you could use the following command to set the `mwmFunctions` resource for the `VendorShell` widget class:

```
XtSetValues mwmFunctions: MWM_FUNC_ALL|MWM_FUNC_RESIZE
```

Unsupported Resources

`dtksh` supports most of the Motif resources. The following lists unsupported resources. Resources with an * (asterisk) can be specified at widget creation time by using `XtSetValues`, but can't be retrieved using `XtGetValues`.

- All widget and gadget Classes:
  - Any fontlist resource *
  - Any pixmap resource *
- Composite:
  - `insertPosition`
  - `children`
- Core:
  - `accelerators`
  - `translations` *
  - `colormap`
- `XmText`:
  - `selectionArray`
  - `selectionArrayCount`
- `ApplicationShell`:
  - `argv`
- `WMShell`:

    - iconWindow
    - windowGroup
- Shell:
    - createPopupChildrenProc
- XmSelectionBox:
    - textAccelerators
- Manager, Primitive, and Gadget Subclasses:
    - userData
- XmFileSelectionBox:
    - dirSearchProc
    - fileSearchProc
    - qualifySearchDataProc

dtksh app-defaults File

The dtksh app-defaults file, named Dtksh, is found in a location based on the following path description:

```
/usr/dt/app-defaults/<LANG>
```

The only information contained in this app-defaults file is the inclusion of the standard Dt base app-defaults file. The following is a listing of the dtksh app-defaults file:

```
#include "Dt"
```

The file Dt is also located in /usr/dt/app-defaults/<LANG> and is shown in the following listing.

```
*foregroundThreshold:70

!###
!#
!#  Help system specific resources
!#
!###

!#
!# Display Area Colors
!#
!#   These resources set the colors for the display area (where
!#   actual help text is displayed). The resources are complex
!#   because they have to override the standard color resources
!#   in all cases.
!#
```

```
*XmDialogShell.DtHelpDialog*DisplayArea.background: White
*XmDialogShell*XmDialogShell.DtHelpDialog*DisplayArea.background:
White
*XmDialogShell.DtHelpDialog*DisplayArea.foreground: Black
*XmDialogShell*XmDialogShell.DtHelpDialog*DisplayArea.foreground:
Black

!#
!# Menu Accelerators
!#
!#   The following resources establish keyboard accelerators
!#   for the most frequently accessed menu commands.
!#

*DtHelpDialogWidget*searchMenu.keyword.acceleratorText: Ctrl+I
*DtHelpDialogWidget*searchMenu.keyword.accelerator: Ctrl<Key>i
*DtHelpDialogWidget*navigateMenu.backTrack.acceleratorText: Ctrl+B
*DtHelpDialogWidget*navigateMenu.backTrack.accelerator: Ctrl<Key>b
*DtHelpDialogWidget*navigateMenu.homeTopic.acceleratorText: Ctrl+H
*DtHelpDialogWidget*navigateMenu.homeTopic.accelerator: Ctrl<Key>h
*DtHelpDialogWidget*fileMenu.close.acceleratorText:   Alt+F4
*DtHelpDialogWidget*fileMenu.close.accelerator: Alt<Key>f4
```

Variable Values

This section describes the types of values for some of the variables in a `dtksh` `app-defaults` file.

Defined Values

The C bindings of the interfaces to X, Xt and Motif include many nonstring values that are defined in header files. The general format of such values consists of an `Xt` or `Xm` prefix followed by a descriptive name. For example, one of the constraint values for a child of a form widget is `XmATTACH_FORM`. Equivalent values are specified in `dtksh` by dropping the prefix, just as in a Motif defaults file:

- `XmDIALOG_COMMAND_TEXT` becomes `DIALOG_COMMAND_TEXT`
- `XtATTACH_FORM` becomes `ATTACH_FORM`

Boolean Values

You can specify a Boolean value as a parameter to a `dtksh` command using the words True or False; case is not significant. A Boolean result is returned as either True or False, using all lowercase letters.

Return Values

Graphical commands in `dtksh` fall into one of four categories, based on the definition of the corresponding C function:

1. The function is void and returns no values. Example: `XtMapWidget()`
2. The function is void, but returns one or more values through reference parameters. Example: `XmGetColors()`
3. The function returns a non-Boolean value. Example: `XtCreateManagedWidget()`
4. The function returns a Boolean value. Example: `XtIsSensitive()`

Category 1

A `dtksh` category 1 command follows the calling sequence of its corresponding C function. The number and order of parameters can be determined by looking at the standard documentation for the function. Example:

```
XtMapWidget $FORM
```

Category 2

A `dtksh` category 2 command also generally follows the calling sequence of its corresponding C function. It returns a value in an environment variable, instead of passing a pointer to a return variable. Example:

```
XmGetColors $FORM $BG FOREGROUND TOPSHADOW BOTTOMSHADOW SELECT
echo "Foreground color = " $FOREGROUND
```

Category 3

A `dtksh` category 3 command differs slightly from its corresponding C function. Where the C function returns its value as the value of the procedure call, a `dtksh` command requires an additional parameter. This parameter is the name of an environment variable into which the return value is to be placed. It is always the first parameter. Example:

```
XmTextGetString TEXT_VALUE $TEXT_WIDGET
echo "The value of the text field is "$TEXT_VALUE
```

Category 4

A `dtksh` category 4 command returns a value that can be used in a conditional expression just as in C. If the C function also returns values through reference variables (as in category 2), the `dtksh` command also uses variable names for the corresponding parameters. Example:

```
if XmIsTraversable $PUSH_BUTTON; then
echo "The pushbutton is traversable"
else
echo "The pushbutton is not traversable"
fi
```

Generally, the order and type of parameters passed to a command matches those passed to the corresponding C function, except as noted for category 3 commands.

Immediate Return Value

Many of the category 3 commands return a single value using an environment variable specified as the first parameter to the command (for these special commands, the first parameter has the name *variable*). If this return value is immediately used in an expression, the special environment variable "−" may be used in place of a variable name. When `dtksh` encounters "−" as the name of the environment variable in which the return value is to be returned, it instead returns the result as the value of the command. This allows the shell script to embed the command call in another command call. This feature only works for commands that return a single value, and the value is returned in the first parameter. For example:

```
XtDisplay DISPLAY $FORM

XSync $DISPLAY true
```

can be replaced by the equivalent statement:

```
XSync $(XtDisplay "-" $FORM) true
```

The reference to `$DISPLAY` is replaced with the value returned by the call to `XtDisplay`.

This capability is available for all category 3 commands except those that create a widget, those that return more than a single value, and those whose first parameter is not a named variable. Commands that do not accept "-" as the environment variable name include the following:

- `XtInitialize()`
- `XtCreateApplicationShell()`
- `XtCreatePopupShell()`
- `XtCreateManagedWidget()`
- `XtCreateWidget()`
- All commands of the form:

  `XmCreate...()`
- Most commands of the form:

  `tt_...()`

Initializing the Xt Intrinsics

A `dtksh` script must first initialize the Xt Intrinsics before it can call any of the Xlib, Xt, Motif, or `libDt` commands. You accomplish this by invoking the `XtInitialize` command, which returns an application shell widget. As is true for all `dtksh` commands that return a widget ID, `XtInitialize` returns the widget ID in the environment variable that is the first argument. For example, in:

```
XtInitialize TOPLEVEL myShellName Dtksh $0 "$@"
```

the widget ID is returned in the environment variable `TOPLEVEL`.

`dtksh` provides a default `app-defaults` file, which is used if the call to `XtInitialize` specifies an application class name of `Dtksh`. This `app-defaults` file contains the standard set of `Dt` application default values, so `dtksh` applications have a consistent look with other `Dt` applications.

Creating Widgets

There are several commands you can use to create widgets:

`XtCreateWidget`	Creates an unmanaged widget.
`XtCreateManagedWidget`	Creates a managed widget.
`XtCreateApplicationShell`	Creates an application shell.
`XtCreatePopupShell`	Creates a pop-up shell.
`XmCreate`<*widgettypes*>	Creates an unmanaged widget.

There is a specific format for each of these commands that you must follow. For example, suppose you want to create an unmanaged push button widget as a child of the top-level widget. You can use either `XtCreateWidget` or `XmCreatePushButton`. The formats for these commands are:

- `XtCreateWidget` *variable name widgetclass $parent [resource:value ...]*
- `XmCreatePushButton` *variable $parent name [resource:value ...]*

The actual commands to create a push button widget are:

```
XtCreateWidget BUTTON button XmPushButton $TOPLEVEL
XmCreatePushButton BUTTON $TOPLEVEL button
```

Each of the preceeding commands do exactly the same thing: create an unmanaged push button. Note that no resource values are set. Suppose that you want the background color of the push button to be red, and the foreground color to be black. You can set the values of these resources this way:

```
XtCreateWidget BUTTON button XmPushButton $TOPLEVEL \
background:Red \
foreground:Black
XmCreatePushButton BUTTON $TOPLEVEL button\
background:Red \
foreground:Black
```

All of the C functions that create a widget return a widget ID, or ID. The corresponding `dtksh` commands set an environment variable equal to the widget ID. These are category 3 commands, so the first argument is the name of the environment variable in which to return the widget ID. The widget ID is an ASCII string used by `dtksh` to access the actual widget pointer. Either of the following commands could be used to create a new form widget; however, in each case the widget ID for the new form widget is returned in the environment variable `FORM`:

- `XtCreateManagedWidget FORM name XmForm $PARENT`
- `XmCreateForm FORM $PARENT name`

After either of these commands, you can use $FORM to reference the new form widget. For example, you could use this command to create a label widget within the new form widget:

```
XmCreateLabel LABEL $FORM name\
labelString:"Hi Mom" \
CH_FORM \
leftAttachment:ATTACH_FORM
```

Note – There is a special widget ID called NULL, provided for cases where a shell script may need to specify a NULL widget. For example, to disable the defaultButton resource for a form widget, use the command

```
XtSetValues $FORM defaultButton:NULL
```

Using a Callback

A callback is a function or procedure that is executed when an event or combination of events occurs. For example, a callback is used to achieve the desired result when a push button is "pressed." It is easy for a dtksh shell script to assign a command to be activated whenever a particular callback is invoked for a widget. The command could be as simple as a string of commands blocked together, or the name of the shell function to invoke.

Registering a Callback

An application registers a callback with a widget to specify a condition in which it is interested and to specify what action should occur when that condition occurs. The callback is registered using XtAddCallback. The action can be any valid dtksh command. For example:

```
XtAddCallback $WIDGET activateCallback "ActivateProc"
XtAddCallback $WIDGET activateCallback \
       "XtSetSensitive $BUTTON false"
```

Passing Data to a Callback

A callback needs to be passed context information, so it can determine what condition led to its call. For a C procedure, this information is typically passed in a callData structure. For example, a scale widget invoking a valueChangedCallback passes an instance of the following structure in callData:

```
typedef struct {
 int reason;
 XEvent event;
 int value;
}XmScaleCallbackStruct;
```

The C application’s callback then does something like:

```
if (scaleCallData->reason == XmCR_VALUE_CHANGED)
{
  eventType = scaleCallData->event->type;
  display = scaleCallData->event->xany.display;
}
```

Similarly, when a callback is invoked in `dtksh`, the following special environment variable is set up before the callback command executes:

`CB_WIDGET`

This is set to the widget ID for the widget that is invoking the callback.

`CB_CALL_DATA`

This is set to the address of the `callData` structure passed by the widget to the callback.

The `CB_CALL_DATA` environment variable represents a pointer to a structure, and access to its fields uses a syntax similar to that of C. Nested environment variables are defined, named the same as the fields of the structure (but all in uppercase), and a dot is used to indicate containment of an element in a structure. Thus, the previous C code to access the `callData` provided by the scale widget translates to:

```
if [ ${CB_CALL_DATA.REASON} = "CR_VALUE_CHANGED" ]; then
  eventType=${CB_CALL_DATA.EVENT.TYPE}
  display=${CB_CALL_DATA.EVENT.XANY.DISPLAY}
fi
```

The same is true of the event structure within the `callData` structure.

For most callback structures, the shell script is able to reference any of the fields defined for the particular callback structure, using the technique described earlier. In most cases, the shell script is not able to alter the values of the fields within these structures. The exception to this is the `XmTextVerifyCallbackStruct`, which is available during the `losingFocusCallback`, the `modifyVerifyCallback` and the `motionVerifyCallback` for the text widget. `dtksh` supports the modification of certain fields within this structure, to the extent that it is supported by Motif. The following fields within the callback structure are capable of being modified:

- `CB_CALL_DATA.DOIT`
- `CB_CALL_DATA.STARTPOS`
- `CB_CALL_DATA.TEXT.PTR`
- `CB_CALL_DATA.TEXT.LENGTH`
- `CB_CALL_DATA.TEXT.FORMAT`

This is an example of how one of these fields can be modified:

- `CB_CALL_DATA.DOIT=”false”`
- `CB_CALL_DATA.TEXT.PTR=”*”`
- `CB_CALL_DATA.TEXT.LENGTH=1`

A Sample Script 2

This chapter shows you how to use what you learned about `dtksh` in Chapter 1. The two simple scripts described here should give you a good start at writing your own scripts.

Writing the Script

This script creates a bulletin board widget within which a push button widget is placed. The script is kept simple by not including any callbacks. The second script includes a callback.

Here's the first script:

```
#!/usr/dt/bin/dtksh
XtInitialize TOPLEVEL dttest1 Dtksh $0
XtSetValues $TOPLEVEL title:"dttest1"
XtCreateManagedWidget BBOARD bboard XmBulletinBoard $TOPLEVEL \
   resizePolicy:RESIZE_NONE height:150 width:250\
   background:SkyBlue
XtCreateManagedWidget BUTTON pushbutton XmPushButton $BBOARD \
   background:goldenrod \
   foreground:MidnightBlue \
   labelString:"Push Here" \
   height:30 width:100 x:75 y:60 shadowThickness:3
XtRealizeWidget $TOPLEVEL
XtMainLoop
```

Figure 2-1 shows the window that the first script produces.

Figure 2-1 Window from script dttest

The first line of the script:

```
#!/usr/dt/bin/dtksh
```

tells the operating system that this script should be executed using `/usr/dt/bin/dtksh` rather than the standard shell.

The next line initializes the Xt Intrinsics.

```
XtInitialize TOPLEVEL dttest1 Dtksh $0
```

The name of the top-level widget is saved in the environment variable `$TOPLEVEL`, the shell widget name is `dttest1`, the application class name is `Dtksh`, and the application name is given by the `dtksh` variable `$0`.

The next line sets the title resource to the name of the script.

```
XtSetValues $TOPLEVEL title:"dttest1"
```

Notice that there is no space between the colon after the resource name (title) and its value. An error message appears if you have a space between them.

The next four lines create a bulletin board widget and set some of its resources.

```
XtCreateManagedWidget BBOARD bboard XmBbulletinBoard $TOPLEVEL \
   resizePolicy:RESIZE_NONE \
   background:SkyBlue\
   height:150 width:250
```

The bulletin board widget's ID is saved in the environment variable `$BBOARD`. The widget's name is `bboard`. This name is used by the Xt Intrinsics to set the values of resources that might be named in an external resource file. The widget class is `XmBulletinBoard`. The bulletin board's parent widget is the widget ID contained in the environment variable `$TOPLEVEL`. This is the toplevel widget created by the initialization command in the first line. The \ (backslash) at the end of the line tells `dtksh` that this command continues on the next line.

The next six lines create a push button widget as a child of the bulletin board, and set some of the push button’s resources.

```
XtCreateManagedWidget BUTTON pushbutton XmPushButton $BBOARD \
   background:goldenrod \
   foreground:MidnightBlue \
   labelString:”Push Here”\
   height:30 width:100 x:75 y:60\
   shadowThickness:3
```

This is basically the same procedure used to create the bulletin board, except that the variable, name, class, and parent are different.

The next line causes the toplevel widget and all its children to be realized.

```
XtRealizeWidget $TOPLEVEL
```

Finally, the `XtMainLoop` command initiates a loop processing of events for the widgets.

```
XtMainLoop
```

In this script, all that happens is the window appears on the display. It stays there until you terminate the script, either by choosing `Close` on the Window Manager menu or by pressing CTRL C in the terminal window from which you executed the script.

Adding a Callback

To provide a function for the push button so that when it is pressed a message appears in the terminal window and the script terminates, you have to add a callback. Also, you must tell the push button about the existence of this callback. The following is the script with the new code added:

```
#!/usr/dt/bin/dtksh

activateCB() {
  echo “Pushbutton activated; normal termination.”
  exit 0
}

XtInitialize TOPLEVEL dttest2 Dtksh $0
```

```
XtSetValues $TOPLEVEL title:"dttest2"
XtCreateManagedWidget BBOARD bboard XmBulletinBoard $TOPLEVEL \
   resizePolicy:RESIZE_NONE \
   background:SkyBlue \
   height:150 width:250
XtCreateManagedWidget BUTTON pushbutton XmPushButton $BBOARD \
   background:goldenrod \
   foreground:MidnightBlue \
   labelString:"Push Here"\
   height:30 width:100 x:75 y:60 shadowThickness:3

XtAddCallback $BUTTON activateCallback activateCB
XtRealizeWidget $TOPLEVEL
XtMainLoop
```

The callback is the function `activateCB()`. You typically add the callback to the push button after it (the push button) has been created:

```
XtAddCallback $BUTTON activateCallback activateCB
```

Now the pushbutton knows about the callback. When you click the push button, the function `activateCB()` is executed, and the message "`Pushbutton activated; normal termination.`" appears in the terminal window from which you executed the script. The script is terminated by the call to the function `exit 0`.

Advanced Topics 3

Now that you have the basic information about `dtksh`, this chapter introduces you to more advanced topics.

Using Context Variables

`dtksh` has a number of variables that provide context to certain aspects of an application.

Event Handler Context Variables

An application registers event handlers with a widget to specify an action to occur when one of the specified events occurs. The action can be any arbitrary `dtksh` command line. For example:

```
XtAddEventHandler $W "Button2MotionMask" false "ActivateProc"
XtAddEventHandler $W "ButtonPressMask|ButtonReleaseMask" \
      false "echo action"
```

Two environment variables are defined to provide context to the event handler:

EH_WIDGET — Set to the ID of the widget for which the event handler is registered.

EH_EVENT — Set to the address of the `XEvent` which triggered the event handler.

Access to the fields within the `XEvent` structure is shown in the following example:

```
if [ ${EH_EVENT.TYPE} = "ButtonPress" ]; then
   echo "X = "${EH_EVENT.XBUTTON.X}
   echo "Y = "${EH_EVENT.XBUTTON.Y}
elif [ ${EH_EVENT.TYPE} = "KeyPress" ]; then
   echo "X = "${EH_EVENT.XKEY.X}
   echo "Y = "${EH_EVENT.XKEY.Y}
fi
```

Translation Context Variables

The Xt Intrinsics provides for event translations to be registered for a widget. Context for event translation is provided in the same way it is provided for event handlers. The two variables defined for translation commands are:

TRANSLATION_WIDGET	Set to the widget handle for the widget for which the translation is registered.
TRANSLATION_EVENT	Set to the address of the `XEvent` that triggered the translation.

Dot-notation provides access to the fields of the event:

```
echo "Event type = "${TRANSLATION_EVENT.TYPE}
echo "Display = "${TRANSLATION_EVENT.XANY.DISPLAY}
```

Workspace Callback Context Variables

An application has the ability to register a callback function that is invoked whenever the user changes to a new workspace. When the callback is invoked, two special environment variables are set, and can be accessed by the shell callback code:

CB_WIDGET	Set to the ID for the widget that is invoking the callback.
CB_CALL_DATA	Set to the X atom that uniquely identifies the new workspace. This can be converted to its string representation, using the `XmGetAtomName` command.

Input Context Variables

The Xt Intrinsics provides the `XtAddInput` facility, which allows an application to register interest in any data available from a particular file descriptor. When programming in C, the application provides a handler function, which is invoked when input is available. It is up to the handler to read the data from the input source and to handle character escaping and line continuations.

`dtksh` also supports the `XtAddInput` facility, but takes it a step further and makes it easier for shell programmers to use. By default, when a shell script registers interest in a file descriptor, `dtksh` invokes the shell script's input handler only when a complete line of text has been received. A complete line of text is defined as a line terminated either by an unescaped newline character or by the end of the file. The input handler is also called if no data is available and the end of the file has been reached. The handler can then use `XtRemoveInput` to remove the input source and to close the file descriptor. The advantage of this default behavior is that input handlers need not be concerned with escape processing or with handling line continuations. The disadvantage is that it assumes that all of the input is line-oriented and contains no binary information.

`dtksh` also supports a "raw" input mode if the input source contains binary information or if the input handler wants to read the data from the input source directly. In raw mode, `dtksh` does not read any of the data from the input source. Whenever `dtksh` is notified that input is available on the input source, it invokes the shell script's input handler. It is then the handler's responsibility to read the incoming data, perform any required buffering and escape processing, and detect when the end of the file has been reached (so that the input source can be removed and the file descriptor closed). This mode seldom needs to be used by a `dtksh` script.

Whether the input handler has been configured to operate in the default mode or in raw mode, `dtksh` sets up several environment variables before calling the shell script's input handler. These environment variables provide the input handler with everything needed to handle the incoming data. The environment variables are:

`INPUT_LINE` — If operating in the default mode, this variable contains the next complete line of input available from the input source. If `INPUT_EOF` is true, then there is no data in this buffer. If operating in raw mode, then this variable always contains an empty string.

`INPUT_EOF` — If operating in the default mode, this variable is set to false anytime `INPUT_LINE` contains data, and it is set to true when the end of file is reached. When the end of file is reached, the shell script's input handler should unregister the input source and close the file descriptor. If operating in raw mode, this variable is always set to false.

`INPUT_SOURCE` — This indicates the file descriptor for which input is available. If operating in raw mode, this file descriptor is used to obtain the pending input. The file descriptor is also used to close the input source, when no longer needed.

`INPUT_ID` — This indicates the ID returned by `XtAddInput`, when the input source was originally registered. This information is needed to remove the input source with `XtRemoveInput`.

Accessing Event Subfields

The `XEvent` structure has many different configurations, based on the event's type. `dtksh` provides access only to the most frequently used `XEvents`. Any of the other standard `XEvents` can be accessed using the event type `XANY`, followed by any of the subfields defined by the `XANY` event structure, which includes the following subfields:

- `${TRANSLATION_EVENT.XANY.TYPE}`
- `${TRANSLATION_EVENT.XANY.SERIAL}`
- `${TRANSLATION_EVENT.XANY.SEND_EVENT}`
- `${TRANSLATION_EVENT.XANY.DISPLAY}`
- `${TRANSLATION_EVENT.XANY.WINDOW}`

`dtksh` supports full access to all of the event fields for the following event types:

- `XANY`
- `XBUTTON`
- `XEXPOSE`
- `XNOEXPOSE`
- `XGRAPHICSEXPOSE`
- `XKEY`
- `XMOTION`

The following examples show how the subfields for the preceding event types can be accessed:

```
${TRANSLATION_EVENT.XBUTTON.X}
$(CB_CALL_DATA.EVENT.XKEY.STATE}
```

```
${EH_EVENT.XGRAPHICSEXPOSE.WIDTH}
```

Responding to a Window Manager Close Notice

When the user selects Close from the Window Manager menu for an application, the application is terminated unless it has arranged to "catch" the Close notification. If the application does not catch the notification, then multiple windows managed by the application all disappear and application data may be left in an undesirable state. To avoid this, `dtksh` provides for catching and handling the Close notification. The application must:

- Define a procedure to handle the Close notification
- Request notification when Close is selected
- Override the response, so the application is not shut down

The following code illustrates this processing.

```
# This is the `callback' invoked when the user selects
# the `Close' menu item
WMCallback()
{
echo "User has selected the Close menu item"
}
# Create the toplevel application shell
XtInitialize TOPLEVEL test Dtksh $0 "$@"
XtDisplay DISPLAY $TOPLEVEL

# Request notification when the user selects the `Close'
# menu item
XmInternAtom DELETE_ATOM $DISPLAY "WM_DELETE_WINDOW" false
XmAddWMProtocolCallback $TOPLEVEL $DELETE_ATOM "WMCallback"

# Ask Motif to not automatically close down your
# application window
XtSetValues $TOPLEVEL deleteResponse:DO_NOTHING
```

Responding to a Session Manager Save State Notice

Session Manager allows applications to save their current state when the user terminates the current session, so that when the user later restarts the session, an application can return to the state it was in. In `dtksh`, this is accomplished by setting up a handler in a similar way of handling a Close notification. If a handler is not set up, the application has to be restarted manually in the new session, and the application does not retain any state.

To set up a handler to save the current state, the application must:

- Define functions to save the state at the end of the session and to restore it on startup
- Register interest in the Session Manager notification
- Register the function to save the state
- At startup, determine whether the saved state should be restored

The following code illustrates this process.

```
#! /usr/dt/bin/dtksh
# Function invoked when the session is being ended by the user
SessionCallback()
{
   # Get the name of the file into which we should save our
   # session information
   if DtSessionSavePath $TOPLEVEL PATH SAVEFILE; then
      exec 9>$PATH

      # Save off whether we are currently in an iconified state
      if DtShellIsIconified $TOPLEVEL ; then
         print -u9 `Iconified'
      else
         print -u9 `Deiconified'
      fi

      # Save off the list of workspaces we currently reside in
      if DtWsmGetWorkspacesOccupied $(XtDisplay "-" $TOPLEVEL) \
            $(XtWindow "-" $TOPLEVEL) \
            CURRENT_WS_LIST ;
      then
         # Map the comma-separated list of atoms into
         # their string representation
         oldIFS=$IFS
         IFS=","
         for item in $CURRENT_WS_LIST;
         do
            XmGetAtomName NAME $(XtDisplay "-" $TOPLEVEL) \
               $item
            print -u9 $NAME
         done
         IFS=$oldIFS
```

```
        fi

        exec 9<&-

        # Let the session manager know how to invoke us when
        # the session is restored
        DtSetStartupCommand $TOPLEVEL \
           "/usr/dt/contrib/dtksh/SessionTest $SAVEFILE"
   else
        echo "DtSessionSavePath FAILED!!"
        exit -3
   fi
}

# Function invoked during a restore session; restores the
# application to its previous state
RestoreSession()
{
   # Retrieve the path where our session file resides
   if DtSessionRestorePath $TOPLEVEL PATH $1; then
        exec 9<$PATH
        read -u9 ICONIFY

        # Extract and restore our iconified state
        case $ICONIFY in
           Iconified) DtSetIconifyHint $TOPLEVEL True;;
           *) DtSetIconifyHint $TOPLEVEL False;
        esac

        # Extract the list of workspaces we belong in, convert
        # them to atoms, and ask the Workspace Manager to relocate
        # us to those workspaces
        WS_LIST=""
        while read -u9 NAME
        do
           XmInternAtom ATOM $(XtDisplay "-" $TOPLEVEL) \
                 $NAME False
           if [ ${#WS_LIST} -gt 0 ]; then
              WS_LIST=$WS_LIST,$ATOM
           else
              WS_LIST=$ATOM
           fi
        done

        DtWsmSetWorkspacesOccupied $(XtDisplay "-" $TOPLEVEL) \
              $(XtWindow "-" $TOPLEVEL) $WS_LIST
```

```
      exec 9<&-
    else
      echo "DtSessionRestorePath FAILED!!"
      exit -3
   fi
}
################## Create the Main UI #######################
XtInitialize TOPLEVEL wmProtTest Dtksh $0 "$@"
XtCreateManagedWidget DA da XmDrawingArea $TOPLEVEL \
                      height:200 width:200
XmInternAtom SAVE_SESSION_ATOM $(XtDisplay "-" $TOPLEVEL) \
      "WM_SAVE_YOURSELF" False

# If a command-line argument was supplied, then treat it as the
# name of the session file
if (( $# > 0))
then
   # Restore to the state specified in the passed-in session file
   XtSetValues $TOPLEVEL mappedWhenManaged:False
   XtRealizeWidget $TOPLEVEL
   XSync $(XtDisplay "-" $TOPLEVEL) False
   RestoreSession $1
   XtSetValues $TOPLEVEL mappedWhenManaged:True
   XtPopup $TOPLEVEL GrabNone
else
   # This is not a session restore, so come up in the default state
   XtRealizeWidget $TOPLEVEL
   XSync $(XtDisplay "-" $TOPLEVEL) False
fi

# Register the fact that we are interested in participating in
# session management
XmAddWMProtocols $TOPLEVEL $SAVE_SESSION_ATOM
XmAddWMProtocolCallback $TOPLEVEL $SAVE_SESSION_ATOM \
         SessionCallback

XtMainLoop
```

Cooperating with Workspace Manager

`dtksh` provides access to all of the major Workspace Manager functions of the Dt libraries, including functions for querying and setting the set of workspaces with which an application is associated; for querying the list of all workspaces; for querying and setting the current workspace; and for requesting that an application be notified any time the user changes to a different workspace.

From a user's perspective, workspaces are identified by a set of names, but from the Workspace Manager's standpoint, workspaces are identified by X atoms. Whenever the shell script asks for a list of workspace identifiers, a string of X atoms is returned. If more than one X atom is present, then the list is comma-separated. The Workspace Manager expects that the shell script uses the same format when passing workspace identifiers back to it. During a given session, it is safe for the shell script to work with the X atoms, since they remain constant over the lifetime of the session. However, as was shown in the Session Manager shell script example in the previous section, if the shell script is going to save and restore workspace identifiers, the identifiers must be converted from their X atom representation to a string before they are saved. Then, when the session is restored, the shell script needs to remap the names into X atoms before passing the information on to the Workspace Manager. Mapping between X atoms and strings, and between strings and X atoms, is accomplished using the following two commands:

- `XmInternAtom ATOM $DISPLAY $WORKSPACE_NAME false`
- `XmGetAtomName NAME $DISPLAY $ATOM`

Specific `dtksh` commands for dealing with workspace management are documented in “Built-in libDt Session Management Commands” in Appendix A.

Creating Localized Shell Scripts

`dtksh` scripts are internationalized and then localized in a process similar to C applications. All strings that may be presented to the user are identified in the script. A post-processor extracts the strings from the script and, from them, builds a catalogue, which can then be translated to any desired locale. When the script executes, the current locale determines which message catalog is searched for strings to display. When a string is to be presented, it is identified by a message-set ID (corresponding to the catalog) and a message number within the set. These values determine what text the user sees. The following code illustrates the process:

```
# Attempt to open our message catalog
catopen MSG_CAT_ID "myCatalog.cat"

# The localized button label is in set 1, and is message # 2
XtCreatePushButton OK $PARENT ok \
  labelString:$(catgets $MSG_CAT_ID 1 2 "OK")

# The localized button label is in set 1, and is message #3
XtCreatePushButton CANCEL $PARENT cancel \
  labelString:$(catgets $MSG_CAT_ID 1 3 "Cancel")

# Close the message catalog, when no longer needed
catclose $MSG_CAT_ID
```

It is important to note that the file descriptor returned by `catopen` must be closed using `catclose` and not by using the kshell `exec` command.

Using dtksh to Access X Drawing Functions

`dtksh` commands include standard Xlib drawing functions to draw lines, points, segments, rectangles, arcs, and polygons. In the standard C programming environment, these functions take a graphics context (GC) as an argument, in addition to the drawing data. In `dtksh` drawing functions, a collection of GC options are specified in the parameter list to the command.

By default, the drawing commands create a GC that is used for that specific command and then discarded. If the script specifies the `-gc` option, the name of a graphics context object can be passed to the command. This GC is used in interpreting the command, and the variable is updated with any modifications to the GC performed by the command.

`-gc` <*GC*> — <*GC*> is the name of an environment variable which has not yet been initialized or which has been left holding a graphic context by a previous drawing command. If this option is specified, then it must be the first GC option specified.

`-foreground` <*color*> — The foreground color, which may be either the name of a color or a pixel number.

`-background` <*color*>	The background color, which may be either the name of a color or a pixel number.
`-font` <*font name*>	The name of the font to be used.
`-line_width` <*number*>	The line width to be used during drawing.
`-function` <*drawing function*>	The drawing function, which can be `xor`, `or`, `clear`, `and`, `copy`, `noop`, `nor`, `nand`, `set`, `invert`, `equiv`, `andReverse`, `orReverse`, or `copyInverted`.
`-line_style` <*style*>	The line style, which can be any of the following: `LineSolid`, `LineDoubleDash`, or `LineOnOffDash`.

Setting Widget Translations

`dtksh` provides mechanisms for augmenting, overriding, and removing widget translations, much as in the C programming environment. In C, an application installs a set of translation action procedures, which can then be attached to specific sequences of events (translations are composed of an event sequence and the associated action procedure). Translations within `dtksh` are handled in a similar fashion, except only a single action procedure is available. This action procedure, named `ksh_eval`, interprets any parameters passed to it as `dtksh` commands and evaluates them when the translation is triggered. The following shell script segment gives an example of how translations can be used:

```
BtnDownProcedure()
{
   echo "Button Down event occurred in button "$1
}
XtCreateManagedWidget BUTTON1 button1 XmPushButton $PARENT \
      labelString:"Button 1" \
      translations:'#augment
            <EnterNotify>:ksh_eval("echo Button1 entered")
            <Btn1Down>:ksh_eval("BtnDownProcedure 1")'
XtCreateManagedWidget BUTTON2 button2 XmPushButton $PARENT \
      labelString:"Button 2"
XtOverrideTranslations $BUTTON2 \
            '#override
            <Btn1Down>:ksh_eval("BtnDownProcedure 2")'
```

A Complex Script 4

This chapter describes a much more complex script than that described in Chapter 2. Because of its length, the entire script is listed in Appendix C. Remember that this guide is not a tutorial on KornShell programming. If you are not familiar with KornShell programming, you should obtain a book on the subject and have it handy for reference.

Using script_find

The script, `script_find`, demonstrates how you can use `dtksh` to provide a graphical interface to the `find` command. `script_find` produces a window within which you can specify parameters for the `find` command. To fully understand the script, you should be familiar with the `find` command and you should have its man page available. A number of the toggle button menu choices in the window produced by `script_find` require some knowledge of the `find` command.

The script's window allows you to specify a search directory and a file name. Other options allow you to place restrictions on the type of file system to search and the file type on which to match. Figure 4-1 shows the script's window.

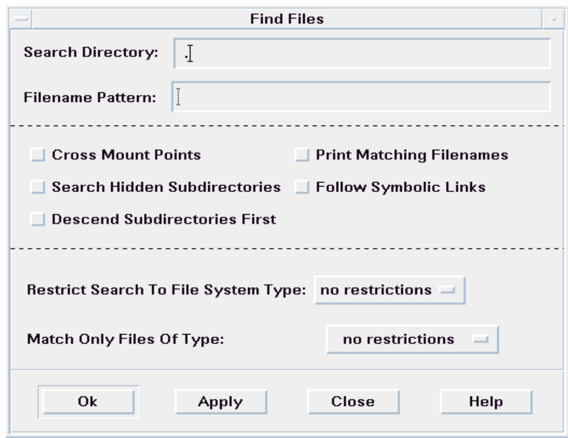

Figure 4-1 Window for script_find

Enter the search directory and file name you're looking for in the text fields at the top of the window. In addition, select any applicable choice (or choices) from the five toggle buttons. You can further restrict the search with the option menus. When you have made all the necessary selections, click OK. If all is well, a window appears shortly thereafter and displays the results of the `find` operation. An error dialog appears if you don't specify a search directory or file name, or if the specified search directory is invalid. For example, suppose you want to find a file called `two_letter_calls`, and you think it resides somewhere in the directory `/users/dlm`. When you enter the directory in the Search Directory text field, you inadvertently type `/users/dln` instead of `/users/dlm`. When you click OK or Apply, `script_find` can't find the directory `/users/dln`, so it creates the error dialog to notify you of this.

Figure 4-2 script_find error dialog

When you correct the mistake, `script_find` then executes properly and creates a `dtterm` window within which it displays the complete path of the file you requested, providing that the file is found.

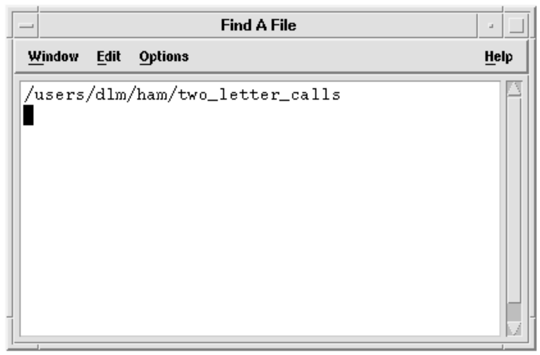

Figure 4-3 Window showing complete path

If `script_find` cannot find the file in the specified directory, nothing appears in the `dtterm` window.

Analyzing script_find

The structure of `script_find` is similar to a C program: some functions and callbacks appear first, followed by the main script.

The first two lines of the script are important, and should be included in every `dtksh` script you write:

```
#! /usr/dt/bin/dtksh
. /usr/dt/lib/dtksh/DtFunc.dtsh
```

The first line executes the `dtksh` system and the second loads the `dtksh` convenience functions. The second line wasn't used in the scripts described in Chapter 2 because those scripts did not use any `dtksh` convenience functions.

Functions and Callbacks

`script_find` has the following functions and callbacks:

- `PostErrorDialog()`
- `OkCallback()`
- `LoadStickyValues()`
- `EvalCmd()`
- `RetrieveAndSaveCurrentValues()`

PostErrorDialog()

This function is called when an error is detected, such as when the user enters an invalid directory. The function calls the convenience function `DtkshDisplayErrorDialog()` which displays a dialog box whose title is Find Error and whose message is contained in the variable $1, which is passed from the calling location.

```
dialogPostErrorDialog()
{
      DtDisplayErrorDialog “Find Error” “$1” \
      DIALOG_PRIMARY_APPLICATION_MODAL
}
```

The last parameter, `DIALOG_PRIMARY_APPLICATION_MODAL`, tells `dtksh` to create a dialog that must be responded to before any other interaction can occur.

OkCallback()

OkCallback() is called when either the OK or Apply button on the main script_find window is pressed. If the OK button is pressed, the script_find window is unmanaged. For either Apply or OK, the input search directory is validated; if it is invalid, then OkCallback() calls PostErrorDialog(). If it is valid, checks are made on the status of the toggle buttons on the script_find window and corresponding adjustments are made to the variable $CMD. This variable contains the entire command that is ultimately executed.

LoadStickyValues()

This function is called from the main program after the window has been created and managed. It loads all the values from the most recent execution of the script. These values are saved in a file called Find.sticky by the function RetrieveandSaveCurrentValues().

EvalCmd()

EvalCmd() is used by LoadStickyValues() to evaluate each line in Find.sticky as a dtksh command. The following is a list of a Find.sticky file:

```
XmTextSetString $SD "/users/dlm"
XmTextFieldSetInsertionPosition $SD 10
XmTextSetString $FNP "two_letter_calls"
XmTextFieldSetInsertionPosition $FNP 16
XtSetValues $FSTYPE menuHistory:$NODIR
XtSetValues $FILETYPE menuHistory:$NOTYPE
XmToggleButtonSetState $T2 true false
XmToggleButtonSetState $T4 true false
```

RetrievAndSaveCurrentValues()

RetrieveAndSaveCurrentValues() retrieves the current settings and values of the widgets in the script_find window and saves them in the file Find.sticky. Find.sticky is then used by LoadStickyValues() the next time the script is executed.

Main Script

The remainder of the script is the equivalent of `Main()` in a C program. It initializes the Xt Intrinsics and creates all the widgets used in the `script_find` window. The `set -f` in the first line tells `dtksh` to suppress expansion of wildcard characters in path names. This is necessary so that the `find` command can perform this expansion.

The `script_find` window (see Figure 4-4) consists of a Form widget with four areas. The areas are marked by Separator widgets, and each area has several widgets, all of which are children of the Form.

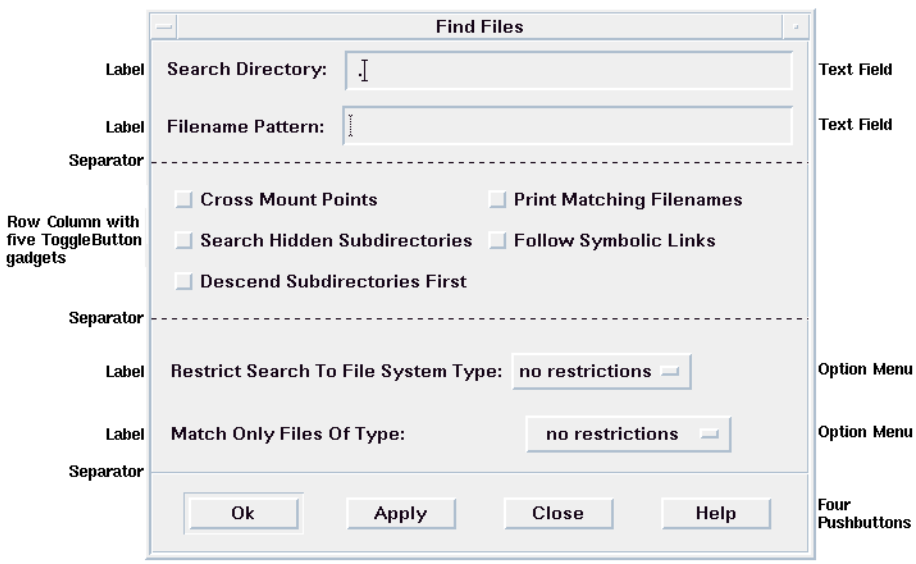

Figure 4-4 Widgets in script_find window

The widgets are created in sequence by area, from top to bottom.

Initialize

Initialize is accomplished by the Xt Intrinsics function `XtInitialize`:

```
XtInitialize TOPLEVEL find Dtksh $0 "${@:-}"
```

This creates a top-level shell that serves as the parent of a Form widget, which is created next.

Create a Form Widget

A Form widget is used as the main parent widget. Form is a Manager widget that allows you to place constraints on its children. Most of the widgets in the main `script_find` window are children of the Form. The description of the creation of the rest of the widgets is separated into the four areas of the window (see Figure 4-4).

First Area

The first area consists of two Label widgets, two TextField widgets, and a Separator widget that separates the first and second areas.

Figure 4-5 First area of `script_find` Window

The following code segment creates and positions the first Label widget and positions it within the Form using the `DtkshAnchorTop` and `DtkshAnchorLeft` convenience functions:

```
XtCreateManagedWidget SDLABEL sdlabel XmLabel $FORM \
   labelString:”Search Directory:” \
   $(DtkshAnchorTop 12) \
   $(DtkshAnchorLeft 10)
```

The following code segment creates and positions the first TextField widget. Note that it is positioned in relation to both the Form and the Label widget.

```
XtCreateManagedWidget SD sd XmText $FORM \
   columns:30 \
   value:”.” \
   $(DtkshAnchorTop 6) \
   $(DtkshRightOf $SDLABEL 10) \
   $(DtkshAnchorRight 10) \
```

```
   navigationType:EXCLUSIVE_TAB_GROUP
XmTextFieldSetInsertionPosition $SD 1
```

The remaining Label widget and TextField widget are created in the same manner.

The Separator widget is created as a child of the Form widget and positioned under the second TextField widget.

```
XtCreateManagedWidget SEP sep XmSeparator $FORM \
   separatorType:SINGLE_DASHED_LINE \
   $(DtkshUnder $FNP 10) \
   $(DtkshSpanWidth)
```

Second Area

The second area consists of a RowColumn widget, five ToggleButton gadgets, and another Separator widget.

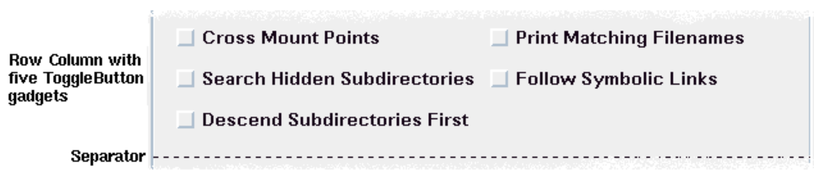

Figure 4-6 Second Area of `script_find` Window

A gadget is a widget that relies on its parent for many of its attributes, thus saving memory resources.

The RowColumn widget is created as a child of the Form widget, and positioned directly under the Separator widget created in the first area.

```
XtCreateManagedWidget RC rc XmRowColumn $FORM \
        orientation:HORIZONTAL \
        numColumns:3 \
        packing:PACK_COLUMN \
   $(DtkshUnder $SEP 10) \
   $(DtkshSpanWidth 10 10) \
   navigationType:EXCLUSIVE_TAB_GROUP
```

The five ToggleButton gadgets are created as children of the RowColumn using the convenience function `DtkshAddButtons`:

```
DtkshAddButtons -w $RC XmToggleButtonGadget \
   T1 "Cross Mount Points"           ""\
   T2 "Print Matching Filenames"     ""\
   T3 "Search Hidden Subdirectories" ""\
   T4 "Follow Symbolic Links"        ""\
   T5 "Descend Subdirectories First" ""
```

Another Separator is then created to separate the second and third areas. Note that this Separator widget ID is called `SEP2`.

```
XtCreateManagedWidget SEP2 sep XmSeparator $FORM \
   separatorType:SINGLE_DASHED_LINE \
   $(DtkshUnder $RC 10) \
   $(DtkshSpanWidth)
```

Third Area

The third area consists of two option menus and another Separator widget.

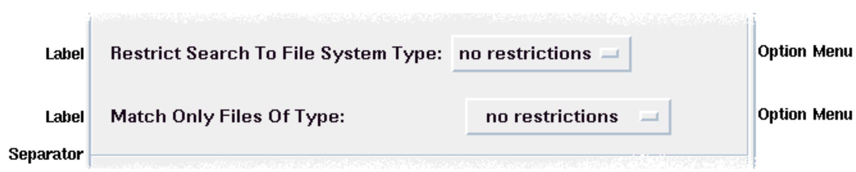

Figure 4-7 Third area of `script_find` Window

The Option Menus are pull-down menus. When the user clicks the option menu button, a menu pane with a number of choices appears. The user drags the pointer to the appropriate choice and releases the mouse button. The menu pane disappears and the option menu button label displays the new choice.

The first option menu pane consists of a number of push button gadgets, representing various restrictions that can be imposed upon the `find` command:

```
XmCreatePulldownMenu PANE $FORM pane
DtkshAddButtons -w $PANE XmPushButtonGadget \
   NODIR "no restrictions" ""\
   NFS   "nfs"             ""\
   CDFS  "cdfs"            ""\
```

```
   HFS   "hfs"             ""
Next, the Option Menu button itself is created and managed, with the
menu pane just created ($PANE) identified as a subMenuId:
XmCreateOptionMenu FSTYPE $FORM fstype \
        labelString:"Restrict Search To File System Type:" \
        menuHistory:$NODIR \
        subMenuId:$PANE \
   $(DtkshUnder $SEP2 20) \
   $(DtkshSpanWidth 10 10) \
   navigationType:EXCLUSIVE_TAB_GROUP
XtManageChild $FSTYPE
```

The second option menu button is created in the same manner. It provides further restrictions on the `find` command.

The third separator is created in the same manner as the other separators.

Fourth Area

The fourth area consists of four push button widgets, all children of the Form widget.

The four push buttons are used as follows:

- OK executes the `find` command with the parameters input in the `script_find` window and removes the `script_find` window.
- Apply executes the `find` command with the parameters input in the `script_find` window but does not remove the `script_find` window.
- Close terminates `script_find` without executing the `find` command.
- `Help` creates a dialog box with information on the use of `script_find`.

The push buttons are created and positioned in much the same manner as any of the other widgets, although they are each labeled differently. The following code segment shows how the OK push button is created:

```
XtCreateManagedWidget OK ok XmPushButton $FORM \
        labelString:"Ok" \
   $(DtkshUnder $SEP3 10) \
```

```
   $(DtkshFloatLeft 4) \
   $(DtkshFloatRight 24) \
   $(DtkshAnchorBottom 10)
XtAddCallback $OK activateCallback "OkCallback"
```

Set Operating Parameters

`XtSetValues` is used to set some initial operating parameters:

```
XtSetValues $FORM \
   initialFocus:$SD \
   defaultButton:$OK \
   cancelButton:$CLOSE \
   navigationType:EXCLUSIVE_TAB_GROUP
```

- Initial focus is set to the first TextField widget in the first area.
- Default button is set to the OK push button in the fourth area.
- Cancel button is set to the Close button in the fourth area.
- Navigation type is set to `EXCLUSIVE_TAB_GROUP`.

The following line configures the TextField widgets so that pressing the return key does not activate the default button within the Form. See the description of `EXCLUSIVE_TAB_GROUP` in Appendix B for more information on its use.

```
DtkshSetReturnKeyControls $SD $FNP $FORM $OK
```

Realize and Loop

The last three lines of the script load the previous values of the `script_find` window, realize the top-level widget, and then enter a loop waiting for user input.

```
LoadStickyValues

XtRealizeWidget $TOPLEVEL
XtMainLoop
```

dtksh Commands A

This appendix contains a list of the commands supported by `dtksh`. Many of these commands are almost identical to their Motif, Xt Intrinsics, or Xlib counterparts. Commands that return a value must have the return variable as an environment variable that is the first parameter in the call. Some commands have more differences.

The following subsections give a synopsis of each of the `dtksh` commands. In general, parameter ordering and types are the same as for corresponding C procedures; exceptions are noted. For more detail on the functionality and parameters of a command, see the standard documentation for the corresponding Xlib, Xt Intrinsics, or Motif procedure.

In the command definitions, parameters named *var*, *var2*, *var3*, and so on, indicate that the shell script should supply the name of an environment variable into which some value will be returned. The word *variable* indicates an environment variable that accepts a return value.

Commands that return a Boolean value (which can be used directly as part of an *if* statement), are noted as such.

Parameters enclosed within `[]` are optional.

Built-in Xlib Commands

`XBell` *display volume*

`XClearArea` *display drawable* [optional GC arguments] *x y width height exposures*

`XClearWindow` *display drawable*

`XCopyArea` *display src dest srcX srcY width height destX destY* [optional GC arguments]

`XDefineCursor` *display window cursor*

`XDrawArc` *display drawable* [optional GC arguments] *x y width height angle1 angle2*

`XDrawLine` *display drawable* [optional GC arguments] *x1 y1 x2 y2*

`XDrawLines` *display drawable* [-*coordinateMode*] [optional GC arguments] *x1 y1 x2 y2 [x3 y3 ...]*

where *coordinateMode* is either `CoordModeOrigin` or `CoordModePrevious`.

`XDrawPoint` *display drawable* [optional GC arguments] *x y*

`XDrawPoints` *display drawable* [-*coordinateMode*] [optional GC arguments] *x1 y1*
[*x2 y2 x3 y3 ...*]

where *coordinateMode* is either `CoordModeOrigin` or `CoordModePrevious`.

`XDrawRectangle` *display drawable* [optional GC arguments] *x y width height*

`XDrawSegments` *display drawable* [optional GC arguments] *x1 y1 x2 y2* [*x3 y3 x4*
y4 ...]

`XDrawString` *display drawable* [optional GC arguments] *x y string*

`XDrawImageString` *display drawable* [optional GC arguments] *x y string*

`XFillArc` *display drawable* [optional GC arguments] *x y width height angle1 angle2*

`XFillPolygon` *display drawable* [-*shape*] [-*coordinateMode*] [optional GC arguments] *x1 y1 x2 y2 ...*

where *shape* is either `Complex`, `Convex`, or `Nonconvex`, and *coordinateMode* is either `CoordModeOrigin` or `CoordModePrevious`.

`XFillRectangle` *display drawable* [optional GC arguments] *x y width height*

`XFlush` *display*

`XHeightOfScreen` *variable screen*

`XRaiseWindow` *display window*

`XRootWindowOfScreen` *variable screen*

`XSync` *display discard*

where *discard* is either `true` or `false`.

`XTextWidth` *variable fontName string*

Note – The `XTextWidth` command is different from the corresponding Xlib procedure because it takes the name of a font instead of a pointer to a font structure.

`XUndefineCursor` *display window*

`XWidthOfScreen` *variable screen*

Built-in Xt Intrinsic Commands

All the Xt Intrinsics commands used to create a new widget require that you specify a widget class for the new widget. The widget (or gadget) class name is the standard class name provided by Motif. For example, the class name for a Motif push button widget is `XmPushButton`, while the class name for the Motif label gadget is `XmLabelGadget`.

`XtAddCallback` *widgetHandle callbackName ksh-command*

where *callbackName* is one of the standard Motif or Xt callback names, with the `Xt` or `Xm` prefix dropped. For example, `activateCallback`.

`XtAddEventHandler` *widgetHandle eventMask nonMaskableFlag ksh-command*

where *eventMask* is of the form *mask*|*mask*|*mask* and the *mask* components are any of the standard set of X event masks, and *nonMaskableFlag* is either `true` or `false`.

`XtAddInput` *variable* [-r] *fileDescriptor ksh-command*

Registers the indicated file descriptor with the X Toolkit as an alternate input source. It is the responsibility of the shell script's input handler to unregister the input source when it is no longer needed and to close the file descriptor.

If the `-r` option is specified (*raw* mode), then `dtksh` does not automatically read any of the data available from the input source; it will be up to the specified kshell command to read all data. If the `-r` option is not specified, then the command specified in *ksh-command* is invoked only when a full line is read (that is, a line terminated by either an unescaped newline character or the end of the file) or when the end of the file is reached. The raw mode is useful for handlers that expect to process nontextual data, or for handlers that do not want `dtksh` automatically reading in a line of data. When the end of file is detected, it is the shell script's input handler's responsibility to use `XtRemoveInput` to remove the input source and to close the file descriptor, if necessary.

In all cases, several environment variables are set up, which can be used by the handler. These include:

`INPUT_LINE`	Empty if in raw mode; otherwise, it contains the next line to be processed.

`INPUT_EOF`	Set to true if end-of-file is reached; otherwise, set to *false*.
`INPUT_SOURCE`	File descriptor associated with this input source.
`INPUT_ID`	The ID associated with this input handler; returned by `XtAddInput`().

`XtAddTimeout` *variable interval ksh-command*

`XtAddWorkProc` *variable ksh-command*

In `dtksh`, the kshell command is typically a kshell function name. Like regular work procedures, this function is expected to return a value that indicates whether the work procedure wants to be called again, or whether it has completed its work and can be automatically unregistered. If the `dtksh` function returns 0, then the work procedure remains registered; any other value causes the work procedure to be automatically unregistered.

`XtAugmentTranslations` *widgetHandle translations*

`XtCreateApplicationShell` *variable applicationName widgetClass* [*resource:value* ...]

`XtCallCallbacks` *widgetHandle callbackName*

where *callbackName* is one of the standard Motif or Xt callback names, with the `Xt` or `Xm` prefix dropped; for example, `activateCallback.`

`XtClass` *variable widgetHandle*

Returns the name of the widget class associated with the passed-in widget handle.

`XtCreateManagedWidget` *variable widgetName widgetClass parentWidgetHandle* [*resource:value* ...]

`XtCreatePopupShell` *variable widgetName widgetClass parentWidgetHandle* [*resource:value* ...]

`XtCreateWidget` *variable widgetName widgetClass parentWidgetHandle* [*resource:value* ...]

`XtDestroyWidget` *widgetHandle* [*widgetHandle* ...]

`XtDisplay` *variable widgetHandle*

`XtDisplayOfObject` *variable widgetHandle*

`XtGetValues` *widgetHandle resource:var1* [*resource:var2* ...]

`XtHasCallbacks` *variable widgetHandle callbackName*

where *callbackName* is one of the standard Motif or Xt callback names, with the `Xt` or `Xm` prefix dropped; for example, `activateCallback`.

variable is set to one of the strings `CallbackNoList`, `CallbackHasNone`, or `CallbackHasSome`.

`XtInitialize` *variable shellName applicationClassName applicationName* [arguments]

Using `Dtksh` as the *applicationClassName* causes the application to use the default `dtksh` `app-defaults` file. The *arguments* parameter is used to reference any command-line arguments that might have been specified by the user of the shell script; these are typically referred to using the shell syntax of "$@".

Returns a value which can be used in a conditional statement.

`XtIsManaged` *widgetHandle*

Returns a value which can be used in a conditional statement.

`XtIsSubclass` *widgetHandle widgetClass*

where *widgetClass* is the name of a widget class. Returns a value which can be used in a conditional statement.

`XtNameToWidget` *variable referenceWidget name*

`XtIsRealized` *widgetHandle*

Returns a value which can be used in a conditional statement.

`XtIsSensitive` *widgetHandle*

Returns a value which can be used in a conditional statement.

`XtIsShell` *widgetHandle*

Returns a value which can be used in a conditional statement.

`XtLastTimestampProcessed` *variable display*

`XtMainLoop`

`XtManageChild` *widgetHandle*

`XtManageChildren` *widgetHandle* [*widgetHandle* ...]

`XtMapWidget` *widgetHandle*

`XtOverrideTranslations` *widgetHandle translations*

`XtParent` *variable widgetHandle*

`XtPopdown` *widgetHandle*

`XtPopup` *widgetHandle grabType*

where *grabType* is one of the strings `GrabNone`, `GrabNonexclusive` or `GrabExclusive`.

`XtRealizeWidget` *widgetHandle*

`XtRemoveAllCallbacks` *widgetHandle callbackName*

where *callbackName* is one of the standard Motif or Xt callback names, with the `Xt` or `Xm` prefix dropped; for example, `activateCallback`

`XtRemoveCallback` *widgetHandle callbackName ksh-command*

where *callbackName* is one of the standard Motif or Xt callback names, with the Xt or Xm prefix dropped; for example, `activateCallback`. As is true with traditional Xt callbacks, when a callback is removed, the same kshell command string must be specified as was specified when the callback was originally registered.

`XtRemoveEventHandler` *widgetHandle eventMask nonMaskableFlag ksh-command*

where *eventMask* is of the form *mask*|*mask*|*mask* and the mask components are any of the standard set of X event masks; that is. *ButtonPressMask* where *nonMaskableFlag* is either *true* or *false.*

As is true with traditional Xt event handlers, when an event handler is removed, the same *eventMask*, *nonMaskableFlag* setting, and kshell command string must be specified as was specified when the event handler was originally registered.

`XtRemoveInput` *inputId*

where *inputId* is the handle that was returned in the specified environment variable when the alternate input source was registered using the `XtAddInput` command.

`XtRemoveTimeOut` *timeoutId*

where *timeoutId* is the handle that was returned in the specified environment variable when the timeout was registered using the `XtAddTimeOut` command.

`XtRemoveWorkProc` *workprocID*

where *workprocID* is the handle that was returned in the specified environment variable when the work procedure was registered using the `XtAddWorkProc` command.

`XtScreen` *variable widgetHandle*

`XtSetSensitive` *widgetHandle state*

where *state* is either *true* or *false.*

`XtSetValues` *widgetHandle resource:value* [*resource:value* ...]

`XtUninstallTranslations` *widgetHandle*

`XtUnmanageChild` *widgetHandle*

`XtUnmanageChildren` *widgetHandle* [*widgetHandle* ...]

`XtUnmapWidget` *widgetHandle*

`XtUnrealizeWidget` *widgetHandle*

`XtWindow` *variable widgetHandle*

Built-in Motif Commands

`XmAddWMProtocolCallback` *widgetHandle protocolAtom ksh-command*

where *protocolAtom* is typically obtained using the `XmInternAtom` command.

`XmAddWMProtocols` *widgetHandle protocolAtom* [*protocolAtom* ...]

where *protocolAtom* is typically obtained using the *XmInternAtom* command.

`XmCommandAppendValue` *widgetHandle string*

`XmCommandError` *widgetHandle errorString*

`XmCommandGetChild` *variable widgetHandle childType*

where `childType` is one of the strings `DIALOG_COMMAND_TEXT`, `DIALOG_PROMPT_LABEL`, `DIALOG_HISTORY_LIST`, or `DIALOG_WORK_AREA`.

`XmCommandSetValue` *widgetHandle commandString*

`XmCreateArrowButton` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateArrowButtonGadget` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateBulletinBoard` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateBulletinBoardDialog` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateCascadeButton` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateCascadeButtonGadget` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateCommand` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateDialogShell` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateDrawingArea` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateDrawnButton` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateErrorDialog` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateFileSelectionBox` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateFileSelectionDialog` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateForm` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateFormDialog` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateFrame` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateInformationDialog` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateLabel` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateLabelGadget` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateList` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateMainWindow` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateMenuBar` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateMenuShell` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateMessageBox` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateMessageDialog` *variable parentWidgetHandle name* *[resource:value ...]*

`XmCreateOptionMenu` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreatePanedWindow` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreatePopupMenu` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreatePromptDialog` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreatePulldownMenu` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreatePushButton` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreatePushButtonGadget` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateQuestionDialog` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateRadioBox` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateRowColumn` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateScale` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateScrollBar` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateScrolledList` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateScrolledText` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateScrolledWindow` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateSelectionBox` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateSelectionDialog` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateSeparator` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateSeparatorGadget` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateText` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateTextField` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateToggleButton` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateToggleButtonGadget` *variable parentWidgetHandle name* *[resource:value* ...]

`XmCreateWarningDialog` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateWorkArea` *variable parentWidgetHandle name* [*resource:value* ...]

`XmCreateWorkingDialog` *variable parentWidgetHandle name* [*resource:value* ...]

`XmFileSelectionDoSearch` *widgetHandle directoryMask*

`XmFileSelectionBoxGetChild` *variable widgetHandle childType*

where *childType* is one of the strings `DIALOG_APPLY_BUTTON`, `DIALOG_CANCEL_BUTTON`, `DIALOG_DEFAULT_BUTTON`, `DIALOG_DIR_LIST`, `DIALOG_DIR_LIST_LABEL`, `DIALOG_FILTER_LABEL`, `DIALOG_FILTER_TEXT`,

DIALOG_HELP_BUTTON, DIALOG_LIST, DIALOG_LIST_LABEL, DIALOG_OK_BUTTON, DIALOG_SEPARATOR, DIALOG_SELECTION_LABEL, DIALOG_TEXT, or DIALOG_WORK_AREA.

`XmGetAtomName` *variable display atom*

`XmGetColors` *widgetHandle background variable var2 var3 var4*

The `XmGetColors` command differs from the C procedure in that it takes a *widgetHandle* instead of a screen pointer and a colormap.

`XmGetFocusWidget` *variable widgetHandle*

`XmGetPostedFromWidget` *variable widgetHandle*

`XmGetTabGroup` *variable widgetHandle*

`XmGetTearOffControl` *variable widgetHandle*

`XmGetVisibility` *variable widgetHandle*

`XmInternAtom` *variable display atomString onlyIfExistsFlag*

where *onlyIfExistsFlag* can be set to either *true* or *false.*

`XmIsTraversable` *widgetHandle*

Returns a value which can be used in a conditional statement.

`XmListAddItem` *widgetHandle position itemString*

The order of the parameters for the `XmListAddItem` command is not identical to its corresponding C programming counterpart.

`XmListAddItems` *widgetHandle position itemString* [*itemString ...*]

The order of the parameters for the `XmListAddItems` command is not identical to its corresponding C programming counterpart.

`XmListAddItemsUnselected` *widgetHandle position itemString [itemString ...]*

The order of the parameters for the `XmListAddItemsUnselected` command is not identical to its corresponding C programming counterpart.

`XmListAddItemUnselected` *widgetHandle position itemString*

The ordering of the parameters to the `XmListAddItemUnselected` command are not identical to its corresponding C programming counterpart.

`XmListDeleteAllItems` *widgetHandle*

`XmListDeleteItem` *widgetHandle itemString*

`XmListDeleteItems` *widgetHandle itemString [itemString ...]*

`XmListDeleteItemsPos` *widgetHandle itemCount position*

`XmListDeletePos` *widgetHandle position*

`XmListDeletePositions` *widgetHandle position [position ...]*

`XmListDeselectAllItems` *widgetHandle*

`XmListDeselectItem` *widgetHandle itemString*

`XmListDeselectPos` *widgetHandle position*

`XmListGetSelectedPos` *variable widgetHandle*

Returns a comma-separated list of indices in *variable*. Returns a value which can be used in a conditional statement.

`XmListGetKbdItemPos` *variable widgetHandle*

`XmListGetMatchPos` *variable widgetHandle itemString*

Returns a comma-separated list of indices in *variable*. Returns a value which can be used in a conditional statement.

`XmListItemExists` *widgetHandle itemString*

Returns a value which can be used in a conditional statement.

`XmListItemPos` *variable widgetHandle itemString*

`XmListPosSelected` *widgetHandle position*

Returns a value which can be used in a conditional statement.

`XmListPosToBounds` *widgetHandle position variable var2 var3 vari4*

Returns a value which can be used in a conditional statement.

`XmListReplaceItemsPos` *widgetHandle position itemString* [*itemString* ...]

The order of the parameters for the `XmListReplaceItemsPos` command is not identical to its corresponding C programming counterpart.

`XmListReplaceItemsPosUnselected` *widgetHandle position itemString [itemString ...]*

The order of the parameters for the `XmListReplaceItemsPosUnselected` command is not identical to its corresponding C programming counterpart.

`XmListSelectItem` *widgetHandle itemString notifyFlag*

where *notifyFlag* can be set to either *true* or *false*.

`XmListSelectPos` *widgetHandle position notifyFlag*

where *notifyFlag* can be set to either *true* or *false*.

`XmListSetAddMode` *widgetHandle state*

where *state* can be set to either *true* or *false*.

`XmListSetBottomItem` *widgetHandle itemString*

`XmListSetBottomPos` *widgetHandle position*

`XmListSetHorizPos` *widgetHandle position*

`XmListSetItem` *widgetHandle itemString*

`XmListSetKbdItemPos` *widgetHandle position*

Returns a value which can be used in a conditional statement.

`XmListSetPos` *widgetHandle position*

`XmListUpdateSelectedList` *widgetHandle*

`XmMainWindowSep1` *variable widgetHandle*

`XmMainWindowSep2` *variable widgetHandle*

`XmMainWindowSep3` *variable widgetHandle*

`XmMainWindowSetAreas` *widgetHandle menuWidgetHandle commandWidgetHandle horizontalScrollbarWidgetHandle verticalScrollbarWidgetHandle workRegionWidgetHandle*

`XmMenuPosition` *widgetHandle eventHandle*

where *eventHandle* refers to an X event, which has typically been obtained by accessing the `CB_CALL_DATA.EVENT`, `EH_EVENT` or `TRANSLATION_EVENT` environment variables.

`XmMessageBoxGetChild` *variable widgetHandle childType*

where *childType* is one of the strings `DIALOG_CANCEL_BUTTON`, `DIALOG_DEFAULT_BUTTON`, `DIALOG_HELP_BUTTON`, `DIALOG_MESSAGE_LABEL`, `DIALOG_OK_BUTTON`, `DIALOG_SEPARATOR`, or `DIALOG_SYMBOL_LABEL`.

`XmOptionButtonGadget` *variable widgetHandle*

`XmOptionLabelGadget` *variable widgetHandle*

`XmProcessTraversal` *widgetHandle direction*

where *direction* is one of the strings `TRAVERSE_CURRENT`, `TRAVERSE_DOWN`, `TRAVERSE_HOME`, `TRAVERSE_LEFT`, `TRAVERSE_NEXT`, `TRAVERSE_NEXT_TAB_GROUP`, `TRAVERSE_PREV`, `TRAVERSE_PREV_TAB_GROUP`, `TRAVERSE_RIGHT`, or `TRAVERSE_UP`.

Returns a value which can be used in a conditional statement.

`XmRemoveWMProtocolCallback` *widgetHandle protocolAtom ksh-command*

where *protocolAtom* is typically obtained using the `XmInternAtom` command.

As is true with traditional Window Manager callbacks, when a callback is removed, the same kshell command string must be specified, as was specified when the callback was originally registered.

`XmRemoveWMProtocols` *widgetHandle protocolAtom* [*protocolAtom* ...]

where *protocolAtom* is typically obtained using the `XmInternAtom` command.

`XmScaleGetValue` *widgetHandle variable*

`XmScaleSetValue` *widgetHandle value*

`XmScrollBarGetValues` *widgetHandle variable var2 var3 var4*

`XmScrollBarSetValues` *widgetHandle value sliderSize increment pageIncrement notifyFlag*

where *notifyFlag* can be set to either *true* or *false*.

`XmScrollVisible` *widgetHandle widgetHandle leftRightMargin topBottomMargin*

`XmSelectionBoxGetChild` *variable widgetHandle childType*

where *childType* is one of the strings `DIALOG_CANCEL_BUTTON`, `DIALOG_DEFAULT_BUTTON`, `DIALOG_HELP_BUTTON`, `DIALOG_APPLY_BUTTON`, `DIALOG_LIST`, `DIALOG_LIST_LABEL`, `DIALOG_OK_BUTTON`, `DIALOG_SELECTION_LABEL`, `DIALOG_SEPARATOR`, `DIALOG_TEXT`, or `DIALOG_WORK_AREA`.

`XmTextClearSelection` *widgetHandle time*

where *time* is typically either obtained from within an X Event or is queried by a call to the `XtLastTimestampProcessed` command.

`XmTextCopy` *widgetHandle time*

where *time* is typically either obtained from within an X Event or is queried by a call to the `XtLastTimestampProcessed` command.

Returns a value which can be used in a conditional statement.

`XmTextCut` *widgetHandle time*

where *time* is typically either obtained from within an X Event or is queried by a call to the `XtLastTimestampProcessed` command.

Returns a value which can be used in a conditional statement.

`XmTextDisableRedisplay` *widgetHandle*

`XmTextEnableDisplay` *widgetHandle*

`XmTextFindString` *widgetHandle startPosition string direction variable*

where *direction* is one of the strings `TEXT_FORWARD` or `TEXT_BACKWARD`.

Returns a value which can be used in a conditional statement.

`XmTextGetBaseline` *variable widgetHandle*

`XmTextGetEditable` *widgetHandle*

Returns a value which can be used in a conditional statement.

`XmTextGetInsertionPosition` *variable widgetHandle*

`XmTextGetLastPosition` *variable widgetHandle*

`XmTextGetMaxLength` *variable widgetHandle*

`XmTextGetSelection` *variable widgetHandle*

`XmTextGetSelectionPosition` *widgetHandle variable var2*

Returns a value which can be used in a conditional statement.

`XmTextGetString` *variable widgetHandle*

`XmTextGetTopCharacter` *variable widgetHandle*

`XmTextInsert` *widgetHandle position string*

`XmTextPaste` *widgetHandle*

Returns a value which can be used in a conditional statement.

`XmTextPosToXY` *widgetHandle position variable var2*

Returns a value which can be used in a conditional statement.

`XmTextRemove` *widgetHandle*

Returns a value which can be used in a conditional statement.

`XmTextReplace` *widgetHandle fromPosition toPosition* string

`XmTextScroll` *widgetHandle lines*

`XmTextSetAddMode` *widgetHandle state*

where *state* can be set to either *true* or *false*.

`XmTextSetEditable` *widgetHandle editableFlag*

where *editableFlag* can be set to either *true* or *false*.

`XmTextSetHighlight` *widgetHandle leftPosition rightPosition* mode

where *mode* is one of the strings `HIGHLIGHT_NORMAL`, `HIGHLIGHT_SELECTED` or `HIGHLIGHT_SECONDARY_SELECTED`.

`XmTextSetInsertionPosition` *widgetHandle position*

`XmTextSetMaxLength` *widgetHandle maxLength*

`XmTextSetSelection` *widgetHandle firstPosition lastPosition time*

where *time* is typically either obtained from within an X Event or is queried by a call to the `XtLastTimestampProcessed` command.

`XmTextSetString` *widgetHandle string*

`XmTextSetTopCharacter` *widgetHandle topCharacterPosition*

`XmTextShowPosition` *widgetHandle position*

`XmTextXYToPos` *variable widgetHandle x y*

`XmTextFieldClearSelection` *widgetHandle time*

where *time* is typically either obtained from within an X Event or is queried by a call to the `XtLastTimestampProcessed` command.

`XmTextFieldGetBaseline` *variable widgetHandle*

`XmTextFieldGetEditable` *widgetHandle*

Returns a value which can be used in a conditional statement.

`XmTextFieldGetInsertionPosition` *variable widgetHandle*

`XmTextFieldGetLastPosition` *variable widgetHandle*

`XmTextFieldGetMaxLength` *variable widgetHandle*

`XmTextFieldGetSelection` *variable widgetHandle*

`XmTextFieldGetSelectionPosition` *widgetHandle variable var2*

Returns a value which can be used in a conditional statement.

`XmTextFieldGetString` *variable widgetHandle*

`XmTextFieldInsert` *widgetHandle position string*

`XmTextFieldPosToXY` *widgetHandle position variable var2*

Returns a value which can be used in a conditional statement.

`XmTextFieldRemove` *widgetHandle*

Returns a value which can be used in a conditional statement.

`XmTextFieldReplace` *widgetHandle fromPosition toPosition string*

`XmTextFieldSetEditable` *widgetHandle editableFlag*

where *editableFlag* can be set to either *true* or *false*.

`XmTextFieldSetHighlight` *widgetHandle leftPosition rightPosition mode*

where *mode* is one of the strings `HIGHLIGHT_NORMAL`, `HIGHLIGHT_SELECTED`, or `HIGHLIGHT_SECONDARY_SELECTED`.

`XmTextFieldSetInsertionPosition` *widgetHandle position*

`XmTextFieldSetMaxLength` *widgetHandle maxLength*

`XmTextFieldSetSelection` *widgetHandle firstPosition lastPosition time*

where *time* is typically either obtained from within an X Event or is queried by a call to the `XtLastTimestampProcessed` command.

`XmTextFieldSetString` *widgetHandle string*

`XmTextFieldShowPosition` *widgetHandle position*

`XmTextFieldXYToPos` *variable widgetHandle x y*

`XmTextFieldCopy` *widgetHandle time*

where *time* is typically either obtained from within an X Event or is queried by a call to the `XtLastTimestampProcessed` command.

Returns a value which can be used in a conditional statement.

`XmTextFieldCut` *widgetHandle time*

where *time* is typically either obtained from within an X Event or is queried by a call to the `XtLastTimestampProcessed` command.

Returns a value which can be used in a conditional statement.

`XmTextFieldPaste` *widgetHandle*

Returns a value which can be used in a conditional statement.

`XmTextFieldSetAddMode` *widgetHandle state*

where *state* can be set to either *true* or *false*.

`XmToggleButtonGadgetGetState` *widgetHandle*

Returns a value which can be used in a conditional statement.

`XmToggleButtonGadgetSetState` *widgetHandle state notifyFlag*

where *state* can be set to either *true* or *false*, and where *notifyFlag* can be set to either *true* or *false*.

`XmToggleButtonGetState` *widgetHandle*

Returns a value which can be used in a conditional statement.

`XmToggleButtonSetState` *widgetHandle state notifyFlag*

where *state* can be set to either *true* or *false*, and where *notifyFlag* can be set to either *true* or *false*.

`XmUpdateDisplay` *widgetHandle*

Built-in TriTeal Enterprise Desktop Application Help Commands

`DtCreateQuickHelpDialog` *variable parentWidgetHandle name* [*resource:value* ...]

`DtCreateHelpDialog` *variable parentWidgetHandle name* [*resource:value* ...]

`DtHelpQuickDialogGetChild` *variable widgetHandle childType*

where *childType* is one of the strings `HELP_QUICK_OK_BUTTON`, `HELP_QUICK_PRINT_BUTTON`, `HELP_QUICK_HELP_BUTTON`, `HELP_QUICK_SEPARATOR`, `HELP_QUICK_MORE_BUTTON`, or `HELP_QUICK_BACK_BUTTON`.

`DtHelpReturnSelectedWidgetId` *variable widgetHandle var2*

variable is set to one of the strings `HELP_SELECT_VALID`, `HELP_SELECT_INVALID`, `HELP_SELECT_ABORT`, or `HELP_SELECT_ERROR`. *var2* is set to the *widgetHandle* for the selected widget.

`DtHelpSetCatalogName` *catalogName*

Built-in Localization Commands

`catopen` *variable catalogName*

Opens the indicated message catalog and returns the catalog ID in the environment variable specified by *variable*. If a shell script needs to close the file descriptor associated with a message catalog, then the catalog ID *must* be closed using the *catclose* command.

`catgets` *variable catalogId setNumber messageNumber defaultMessageString*

Attempts to extract the requested message string from the message catalog associated with the *catalogId* parameter. If the message string cannot be located, then the default message string is returned. In either case, the returned message string is placed into the environment variable indicated by *variable*.

`catclose` *catalogId*

Closes the message catalog associated with the indicated *catalogId*.

Built-in libDt Session Management Commands

`DtSessionRestorePath` *widgetHandle variable sessionFile*

Given the file name for the session file (excluding any path information), this command returns the full path for the session file in the environment variable *variable*.

Returns 0 if successful, 1 if unsuccessful.

`DtSessionSavePath` *widgetHandle variable var2*

The full path name for the session file is returned in the environment variable *variable*. The file name portion of the session file (excluding any path information) is returned in the environment variable indicated by *var2*.

Returns 0 if successful, 1 if unsuccessful.

`DtShellIsIconified` *widgetHandle*

Allows a shell script to query the iconified state of a shell window. Returns 0 if successful, 1 if unsuccessful.

`DtSetStartupCommand` *widgetHandle commandString*

Part of the session management process is telling the Session Manager how to restart your application the next time the user reopens the session. This command passes the specified command string to the Session Manager. The widget handle should refer to an application shell.

`DtSetIconifyHint` *widgetHandle iconifyHint*

where *iconifyHint* can be set to either true or false.

Allows the initial iconified state for a shell window to be set. This command only works if the window associated with the widget has been realized but not yet displayed.

Built-in libDt Workspace Management Commands

`DtWsmAddCurrentWorkspaceCallback` *variable widgetHandle ksh-command*

Evaluates the specified kshell command whenever the user changes workspaces. The handle associated with this callback is returned in the environment variable indicated by *variable*. The widget indicated by *widgetHandle* should be a shell widget.

`DtWsmRemoveWorkspaceCallback` *callbackHandle*

Removes a workspace notification callback. When removing a workspace callback, you must pass in the callback handle that was returned when you registered the callback with *DtWsmAddCurrentWorkspaceCallback*.

`DtWsmGetCurrentWorkspace` *display rootWindow variable*

Returns the X atom that represents the user's current workspace in the environment variable indicated by *variable*. Use the `XmGetAtomName` command to map the X atom into its string representation.

`DtWsmSetCurrentWorkspace` *widgetHandle workspaceNameAtom*

Changes the user's current workspace to the workspace indicated by *workspaceNameAtom*.

Returns 0 if successful, 1 if unsuccessful.

`DtWsmGetWorkspaceList` *display rootWindow variable*

Returns a string of comma-separated X atoms, representing the current set of workspaces defined for the user, in the environment variable indicated by *variable*.

Returns 0 if successful, 1 if unsuccessful.

`DtWsmGetWorkspacesOccupied` *display window variable*

Returns a string of comma-separated X atoms, representing the current set of workspaces occupied by the indicated shell window in the environment variable indicated by *variable*.

Returns 0 if successful, 1 if unsuccessful.

`DtWsmSetWorkspacesOccupied` *display window workspaceList*

Moves the indicated shell window to the set of workspaces indicated by the string *workspaceList*, which must be a comma-separated list of X atoms.

`DtWsmAddWorkspaceFunctions` *display window*

Forces the Window Manager menu to include the functions used to move the window to other workspaces. This command only works if the window is in the withdrawn state.

`DtWsmRemoveWorkspaceFunctions` *display window*

Forces the Window Manager menu to not display the functions used to move the window to other workspaces; this prevents the window from being moved to any other workspaces. This command only works if the window is in the withdrawn state.

`DtWsmOccupyAllWorkspaces` *display window*

Requests that a window occupy all workspaces, including new workspaces, as they are created.

`DtWsmGetCurrentBackdropWindows` *display rootWindow variable*

Returns a string of comma-separated window IDs, representing the set of root backdrop windows.

Built-in libDt Action Commands

The set of commands in this section provide you with the tools for loading the action databases, querying information about actions defined in the databases, and requesting that an action be initiated.

`DtDbLoad`

Reads in the action and data-types databases. If called multiple times, then the old databases are freed before the new ones are read. This command must be called before any of the other `libDt` action commands, or any of the `libDt` data typing commands. The shell script should also use the *DtDbReloadNotify* command, so that the shell script can be notified if new databases must be loaded.

`DtDbReloadNotify` *ksh-command*

Requests notification whenever the action or data-types databases need to be reloaded. The specified kshell command is executed when the notification is received. Typically, the kshell command includes a call to the `DtDbLoad` command.

Note – The above command needs to have an X connection in order to work (i.e. must have called XtInitialize).

`DtActionExists` *actionName*

Tests to see if an action exists in the database with the name specified by the *actionName* parameter. Returns a value which can be used in a conditional statement.

Note – The above command needs to have an X connection in order to work (i.e. must have called XtInitialize).

`DtActionLabel` *variable actionName*

Returns the localizable LABEL attribute associated with the indicated action. If the action does not exist, then an empty string is returned.

`DtActionDescription` *variable actionName*

Returns the value of the DESCRIPTION attribute associated with the indicated action. An empty string is returned if the action is not defined, or if the DESCRIPTION attribute was not specified.

Built-in libDt Data-Typing Commands

`DtLoadDataTypes`

Loads the data-typing databases and should be invoked before any of the other data-typing commands.

`DtDtsFileToDataType` *variable filePath*

Returns the name of the data type associated with the file indicated by the *filePath* argument in the *variable* argument. The *variable* argument is set to an empty string if the file cannot be typed.

`DtDtsFileToAttributeValue` *variable filePath attrName*

Returns the string representing the value of the specified attribute for the data type associated with the indicated file. If the attribute is not defined, or if the file could not be typed, then the *variable* argument is set to an empty string.

`DtDtsFileToAttributeList` *variable filePath*

Returns the space-separated list of attribute names defined for the data type associated with the indicated file. A shell script can then query the individual values for the attributes, using the `DtDtsFileToAttributeValue` command. The *variable* argument is set to an empty string if the file cannot be typed. This command differs from its corresponding C programming counterpart, in that it only returns the names of the defined attributes and not their values.

`DtDtsDataTypeToAttributeValue` *variable dataType attrName optName*

Returns the string representing the value of the specified attribute for the indicated data type. If the attribute is not defined, or if the indicated data type does not exist, then the *variable* argument is set to an empty string.

`DtDtsDataTypeToAttributeList` *variable dataType optName*

Returns the space-separated list of attribute names defined for the indicated data type. A shell script can then query the individual values for the attributes, using the `DtDtsDataTypeToAttributeValue` command. The *variable* argument is set to an empty string if the data type is not defined. This command differs from its corresponding C programming counterpart, in that it only returns the names of the defined attributes and not their values.

`DtDtsFindAttribute` *variable name value*

Returns a space-separated list of datatype names whose attribute indicated by the *name* argument has the value indicated by the *value* argument. If an error occurs, the *variable* argument is set to an empty string.

`DtDtsDataTypeNames` *variable*

Returns a space-separated list representing all the data types currently defined in the data-types database. If an error occurs, then the *variable* argument is set to an empty string.

`DtDtsSetDataType` *variable filePath dataType override*

Sets a data type for the specified directory. The *variable* argument is set to the resultant saved data type for the directory.

`DtDtsDataTypeIsAction` *dataType*

Determines whether a particular data type represents an action entry. Returns a value which can be used in a conditional statement.

Miscellaneous Built-in libDt Commands

`_DtGetHourGlassCursor` *variable display*

Returns the X cursor ID associated with the standard `Dt` hourglass cursor.

`_DtTurnOnHourGlass` *widgetHandle*

Turns on the standard `Dt` hourglass cursor for the indicated widget.

`_DtTurnOffHourGlass` *widgetHandle*

Turns off the standard `Dt` hourglass cursor for the indicated widget.

Built-in Desktop Services Message Set Commands

The following set of commands implements the minimum subset of the Desktop Services Message Set required to allow a shell script to participate in the Desktop Services protocol. Many of the ToolTalk commands differ slightly from their associated C programming call. For ToolTalk commands that typically return a pointer, a C application validates that pointer by calling the `tt_ptr_error`() function; this function call returns a `Tt_status` value, which indicates whether the pointer was valid, and if not, why it was not valid. Because of the kshell code's design, the string pointer that the shell script sees is not typically the same as the string pointer returned by the underlying C code. Typically, during shell programming, this is not a problem because the important information is the string value, not the string pointer.

To allow shell scripts to get the status of a pointer, any of the commands that normally return a pointer also return the associated `Tt_status` value for the pointer automatically. This saves the shell script from needing to make an additional call to check the validity of the original pointer. In the case of a pointer error occurring, `dtksh` returns an empty string for the pointer value and sets the `Tt_status` code accordingly.

The `Tt_status` value is returned in the *status* argument. The `Tt_status` value is a string representing the error and can assume any of the following values:

```
TT_OK
TT_WRN_NOTFOUND
TT_WRN_STALE_OBJID
TT_WRN_STOPPED
TT_WRN_SAME_OBJID
TT_WRN_START_MESSAGE
TT_ERR_CLASS
TT_ERR_DBAVAIL
TT_ERR_DBEXIST
TT_ERR_FILE
TT_ERR_INVALID
TT_ERR_MODE
TT_ERR_ACCESS
TT_ERR_NOMP
TT_ERR_NOTHANDLER
TT_ERR_NUM
TT_ERR_OBJID
TT_ERR_OP
TT_ERR_OTYPE
TT_ERR_ADDRESS
TT_ERR_PATH
TT_ERR_POINTER
TT_ERR_PROCID
TT_ERR_PROPLEN
TT_ERR_PROPNAME
TT_ERR_PTYPE
TT_ERR_DISPOSITION
TT_ERR_SCOPE
TT_ERR_SESSION
TT_ERR_VTYPE
TT_ERR_NO_VALUE
TT_ERR_INTERNAL
TT_ERR_READONLY
```

TT_ERR_NO_MATCH
TT_ERR_UNIMP
TT_ERR_OVERFLOW
TT_ERR_PTPE_START
TT_ERR_CATEGORY
TT_ERR_DBUPDATE
TT_ERR_DBFULL
TT_ERR_DBCONSIST
TT_ERR_STATE
TT_ERR_NOMEM
TT_ERR_SLOTNAME
TT_ERR_XDR
TT_DESKTOP_EPERM
TT_DESKTOP_ENOENT
TT_DESKTOP_EINTR
TT_DESKTOP_EIO
TT_DESKTOP_EAGAIN
TT_DESKTOP_ENOMEM
TT_DESKTOP_EACCES
TT_DESKTOP_EFAULT
TT_DESKTOP_EEXIST
TT_DESKTOP_ENODEV
TT_DESKTOP_ENOTDIR
TT_DESKTOP_EISDIR
TT_DESKTOP_EINVAL
TT_DESKTOP_ENFILE
TT_DESKTOP_EMFILE
TT_DESKTOP_ETXBSY
TT_DESKTOP_EFBIG
TT_DESKTOP_ENOSPC
TT_DESKTOP_EROFS
TT_DESKTOP_EMLINK
TT_DESKTOP_EPIPE
TT_DESKTOP_ENOMSG
TT_DESKTOP_EDEADLK
TT_DESKTOP_ECANCELED
TT_DESKTOP_ENOTSUP
TT_DESKTOP_ENODATA
TT_DESKTOP_EPROTO
TT_DESKTOP_ENOTEMPTY

```
TT_DESKTOP_ETIMEDOUT
TT_DESKTOP_EALREADY
TT_DESKTOP_UNMODIFIED
TT_MEDIA_ERR_SIZE
TT_MEDIA_ERR_FORMAT
```

Some of the commands take a message scope as a parameter. The scope indicates which clients have the potential of receiving the outgoing message. For these commands, the *scope* parameter can be set to any of the following values:

```
TT_SCOPE_NONE
TT_SESSION
TT_FILE
TT_BOTH
TT_FILE_IN_SESSION
```

`tt_file_netfile` *variable status filename*

Converts the indicated *filename*, assumed to be a valid file name on the local host, to its corresponding *netfilename* format. A *netfilename* can be passed to other hosts on a network and then converted back to a path relative to the other host, using the `tt_netfile_file` command.

`tt_netfile_file` *variable status netfilename*

Converts the indicated *netfilename* to a path name that is valid on the local host.

`tt_host_file_netfile` *variable status host filename*

Converts the indicated file, assumed to be resident on the specified host, into its corresponding *netfilename* format.

`tt_host_netfile_file` *variable status host netfilename*

Converts the indicated *netfilename* into a valid path on the indicated host.

`ttdt_open` *variable status var2 toolname vendor version sendStarted*

Opens a ToolTalk communications endpoint. It returns in the *variable* argument the procID associated with this connection. It returns the file descriptor associated with this connection in *var2*; this file descriptor can be used to register an alternate Xt input handler. The *sendStarted* argument is a value and if set to *true*, causes a `Started` message to be automatically sent.

Any procIDs returned by `ttdt_open` contain embedded spaces. To prevent kshell from interpreting the procID as a multiple parameter (versus a single parameter with embedded spaces), you should always enclose any references to the environment variable containing the procID within double quotes, as shown:

```
ttdt_close STATUS "$PROC_ID" "" True
```

`tttk_Xt_input_handler` *procID source id*

For the ToolTalk messages to be received and processed, the shell script must register an Xt input handler for the file descriptor returned by the call to `ttdt_open`. The Xt input handler is registered using the `XtAddInput` command, and the handler *must* be registered as a *raw* input handler. The input handler that the shell script registers should invoke `tttk_Xt_input_handler` to get the message received and processed. The following code block demonstrates how this is done:

```
ttdt_open PROC_ID STATUS FID "Tool" "HP" "1.0" True \
  XtAddInput INPUT_ID -r $FID "ProcessTTInput \"$PROC_ID\""
ProcessTTInput()
{
  tttk_Xt_input_handler $1 $INPUT_SOURCE $INPUT_ID
}
```

`$INPUT_SOURCE` is set by `XtAddInput` to indicate the file description that activated this input process.

Refer to the description of the `XtAddInput` command for more details about alternate Xt input handlers.

Note that the `\"` (backslash and double quotation mark) characters before and after the reference to the procID environment variable are necessary, because the value contained in the procID environment variable contains embedded spaces and could be misinterpreted unless escaped as shown.

`ttdt_close` *status procID newProcId sendStopped*

Closes the indicated communications connection and optionally sends a *Stopped* notice, if the *sendStopped* argument is set to *true*.

Because the procID returned by the call to `ttdt_open` contains embedded spaces, it is necessary to enclose any references to the procID environment variable within double quotation marks:

```
ttdt_close STATUS "$PROC_ID" "$NEW_PROC_ID" False
```

`ttdt_session_join` ***variable*** *status sessId shellWidgetHandle join*

Joins the session indicated by the *sessId* argument as a good desktop citizen, by registering patterns and default callbacks for many standard desktop message interfaces. If the *sessId* argument does not specify a value (that is, it is an empty string), then the default session is joined. If the *shellWidgetHandle* argument specifies a widget handle (that is, it is not an empty string), then it should refer to a `mappedWhenManaged applicationShellWidget`. The *join* argument is a Boolean and should be set to true or false. This command returns an opaque *pattern* handle in the *variable* argument; when no longer needed, this handle can be destroyed using the `ttdt_session_quit` command.

`ttdt_session_quit` *status sessId sessPatterns quit*

Destroys the message patterns specified by the *sessPatterns* argument and, if the *quit* argument is set to *true*, quits the session indicated by the *sessId* argument or quits the default session if *sessId* is empty.

`ttdt_file_join` *variable status pathName scope join ksh-command*

Registers interest in the deleted, modified, reverted, moved, and saved messages for the indicated file in the indicated scope. An opaque pattern handle is returned in the *variable* argument. When no longer interested in monitoring messages for the indicated file, this should be destroyed by calling `ttdt_file_quit`.

The requested *ksh-command* is evaluated anytime one of the messages is received for the indicated file. When this kshell command is evaluated, the following environment variables are defined and provide additional information about the received message:

`DT_TT_MSG`	Contains the opaque handle for the incoming message

`DT_TT_OP` Contains the string representing the operation to be performed; that is, `TTDT_DELETED`, `TTDT_MODIFIED`, `TTDT_REVERTED`, `TTDT_MOVED` or `TTDT_SAVED`.

`DT_TT_PATHNAME` Contains the pathname for the file to which this message pertains.

`DT_TT_SAME_EUID_EGID` Set to True if the message was sent by an application operating with the same effective user ID (euid) and effective group ID (egid) as this process.

`DT_TT_SAME_PROCID` Set to True if the message was sent by an application with the same procID (as returned by `ttdt_open`).

When the callback completes, it *must* indicate whether the passed-in message was "consumed" (replied-to, failed, or rejected). If the callback returns the message (as passed-in in the `DT_TT_MSG` environment variable), then it is assumed that the message was not consumed. If the message was consumed, then the callback should return 0, or one of the values returned by the `tt_error_pointer` command. The callback can return its value in the following fashion:

```
return $DT_TT_MSG (or) return 0
```

`ttdt_file_quit` *status patterns quit*

Destroys the message patterns specified by the *patterns* argument and unregisters interest in the path name that was passed to the *ttdt_file_join* command, if *quit* is set to *true*. The *patterns* argument should be the value that was returned by the call to the `ttdt_file_join` command.

`ttdt_file_event` *status op patterns send*

Creates, and optionally sends, a ToolTalk notice announcing an event pertaining to a file. The file is indicated by the path name that was passed to the `ttdt_file_join` command when *patterns* was created. The *op* argument indicates what should be announced for the indicated file, and it can be set to `TTDT_MODIFIED`, `TTDT_SAVED`, or `TTDT_REVERTED`. If *op* is set to `TTDT_MODIFIED`, then this command registers to handle *Get_Modified*, *Save* and *Revert* messages in the scope specified when the *patterns* were created. If *op* is set to TTDT_SAVED or TTDT_REVERTED, this command unregisters from handling *Get_Modified*, *Save,* and *Revert* messages for this file. If the *send* argument is set to *true*, then the indicated message is sent.

ttdt_Get_Modified *pathName scope timeout*

Sends a *Get_Modified* request in the indicated scope and waits for a reply or for the specified timeout (in milliseconds) to elapse. A *Get_Modified* request asks other ToolTalk clients if they have any changes pending on *pathname* that they intend to make persistent. Returns a value which can be used in a conditional statement. A value of *true* is returned if an affirmative reply is received within the specified timeout; otherwise, false is returned.

ttdt_Save *status pathName scope timeout*

Sends a *Save* request in the indicated *scope* and waits for a reply or for the indicated *timeout* (in milliseconds) to elapse. A *Save* request asks the handling ToolTalk client to save any changes pending for the file specified in the *pathName* argument. A status of TT_OK is returned if an affirmative reply is received before the timeout elapses. Otherwise, one of the standard Tt_status error values is returned.

ttdt_Revert *status pathName scope timeout*

Sends a *Revert* request in the indicated *scope* and waits for a reply or for the indicated *timeout* (in milliseconds) to elapse. A *Revert* request asks the handling ToolTalk client to discard any changes pending for the file specified in the *pathName* argument. A status of TT_OK is returned if an affirmative reply is received before the timeout elapses. Otherwise, one of the standard *Tt_status* error values is returned.

The following commands are typically used by the callback registered with the ttdt_file_join command. They serve as the mechanism for consuming and destroying a message. A message is *consumed* by either rejecting, failing, or replying to it. *tt_error_pointer* can be used by the callback to obtain a return pointer for indicating an error condition.

tt_error_pointer *variable ttStatus*

Returns a “magic value,” which is used by ToolTalk to represent an invalid pointer. The magic value returned depends upon the *ttStatus* value passed-in. Any of the valid Tt_status values may be specified.

`tttk_message_destroy` *status msg*

Destroys any patterns that may have been stored on the message indicated by the *msg* argument, and then destroys the message.

`tttk_message_reject` *status msg msgStatus msgStatusString destroy*

Sets the status and the status string for the indicated request message, and then rejects the message. It then destroys the passed-in message, if the *destroy* argument is set to `True`. This command is one way in which the callback specified with the `ttdt_file_join` command can consume a message. It is typically safe to destroy the message, using `tttk_message_destroy`, after rejecting the message.

`tttk_message_fail` *status msg msgStatus msgStatusString destroy*

Sets the status and the status string for the indicated request message, and then fails the message. It then destroys the passed-in message, if the *destroy* argument is set to `True`. This command is one way in which the callback specified with the `ttdt_file_join` command can consume a message. It is typically safe to destroy the message, using `tttk_message_destroy`, after failing the message.

`tt_message_reply` *status msg*

Informs the ToolTalk service that the shell script has handled the message and filled in all return values. The ToolTalk service then sends the reply back to the sending process, filling in the state as `TT_HANDLED`. After replying to a message, it is typically safe to destroy the message, using the `tttk_message_destroy` command.

dtksh Convenience Functions B

The dtksh utility includes a file of convenience functions. This file is itself a shell script containing shell functions that may be useful to a shell programmer. The shell functions perform operations that dtksh programmers frequently have to do for themselves. These include functions for quickly creating certain kinds of dialogs (help, error, warning, and so on), a function for easily creating a collection of buttons, and functions that make it easier to configure the constraint resources for a child of a form widget. It is not a requirement that shell script writers use these convenience functions; they are supplied to make it easier for developers to write shorter and more readable shell scripts.

Before a shell script can access these functions, it must first include the file containing the convenience functions. The convenience functions are located in the file /usr/dt/lib/dtksh/DtFuncs.dtsh. Use the following notation to include them in a shell script:

```
. /usr/dt/lib/dtksh/DtFuncs.dtsh
```

DtkshAddButtons

`DtkshAddButtons` adds one or more buttons of the same kind into a composite widget. It is most often used to add a collection of buttons into a menupane or menubar.

Usage:

```
DtkshAddButtons parent widgetClass label1 callback1
                [label2 callback2 ...]
DtkshAddButtons [-w] parent widgetClass variable1 label1 callback1 \
                [variable2 label2 callback2 ...]
```

The `-w` option indicates that the convenience function should return the widget handle for each of the buttons it creates. The widget handle is returned in the specified environment variable. The `widgetClass` parameter can be set to any of the following, but it defaults to `XmPushButtonGadget` if nothing is specified.

- `XmPushButton`
- `XmPushButtonGadget`
- `XmToggleButton`
- `XmToggleButtonGadget`
- `XmCascadeButton`
- `XmCascadeButtonGadget`

Examples:

```
DtkshAddButtons $MENU XmPushButtonGadget Open do_Open Save do_Save
                Quit exit
DtkshAddButtons -w $MENU XmPushButtonGadget B1 Open do_Open B2 Save
                do_Save
```

DtkshSetReturnKeyControls

`DtkshSetReturnKeyControls` configures a text widget within a form widget so that the Return key does not activate the default button within the form, but instead moves the focus to the next text widget within the form. This is useful if you have a window that contains a series of text widgets, and the default button should not be activated until the user presses the Return key while the focus is in the last text widget.

Usage:

```
DtkshSetReturnKeyControls textWidget nextTextWidget formWidget
                          defaultButton
```

The *textWidget* parameter specifies the widget to be configured to catch the Return key and force the focus to move to the next text widget (as indicated by the *nextTextWidget* parameter). The *formWidget* parameter specifies the form containing the default button and should be the parent of the two text widgets. The *defaultButton* parameter indicates which component is to be treated as the default button within the form widget.

Examples:

```
DtkshSetReturnKeyControls $TEXT1 $TEXT2 $FORM $OK
DtkshSetReturnKeyControls $TEXT2 $TEXT3 $FORM $OK
```

DtkshUnder, DtkshOver, DtkshRightOf, and DtkshLeftOf

These convenience functions simplify the specification of certain classes of form constraints. They provide a way of attaching a component to one edge of another component. They are used when constructing the resource list for a widget. This behavior is accomplished using the `ATTACH_WIDGET` constraint.

Usage:

```
DtkshUnder widgetId [offset]
```

```
DtkshOver widgetId [offset]
```

```
DtkshRightOf widgetId [offset]
```

```
DtkshLeftOf widgetId [offset]
```

The *widgetId* parameter specifies the widget to which the current component is to be attached. The *offset* value is optional and defaults to 0 if not specified.

Example:

```
XtCreateManagedWidget BUTTON4 button4 XmPushButton $FORM \
        labelString:"Exit" \
        $(DtkshUnder $BUTTON2) \
        $(DtkshRightOf $BUTTON3)
```

DtkshFloatRight, DtkshFloatLeft, DtkshFloatTop, and DtkshFloatBottom

These convenience functions simplify the specification of certain classes of form constraints. They provide a way of positioning a component, independent of the other components within the form. As the form grows or shrinks, the component maintains its relative position within the form. The component may still grow or shrink, depending upon the other form constraints specified for the component. This behavior is accomplished using the `ATTACH_POSITION` constraint.

Usage:

```
DtkshFloatRight [position]

DtkshFloatLeft [position]

DtkshFloatTop [position]

DtkshFloatBottom [position]
```

The optional *position* parameter specifies the relative position to which the indicated edge of the component is positioned. The *position* value is optional and defaults to 0 if one is not specified.

Example:

```
XtCreateManagedWidget BUTTON1 button1 XmPushButton $FORM \
        labelString:"Ok" \
        $(DtkshUnder $SEPARATOR) \
        $(DtkshFloatLeft 10) \
        $(DtkshFloatRight 40)
```

DtkshAnchorRight, DtkshAnchorLeft, DtkshAnchorTop, and DtkshAnchorBottom

These convenience functions simplify the specification of certain classes of form constraints. They provide a way of attaching a component to one of the edges of a form widget in such a way that, as the form grows or shrinks, the component's position does not change. However, depending upon the other form constraints set on this component, it may still grow or shrink in size. This behavior is accomplished using the `ATTACH_FORM` constraint.

Usage:

```
DtkshAnchorRight [offset]

DtkshAnchorLeft [offset]

DtkshAnchorTop [offset]

DtkshAnchorBottom [offset]
```

The optional *offset* parameter specifies how far from the edge of the form widget the component should be positioned. If an offset is not specified, then 0 is used.

Example:

```
XtCreateManagedWidget BUTTON1 button1 XmPushButton $FORM \
        labelString:"Ok" \
        $(DtkshUnder $SEPARATOR) \
        $(DtkshAnchorLeft 10) \
        $(DtkshAnchorBottom 10)
```

DtkshSpanWidth and DtkshSpanHeight

These convenience functions simplify the specification of certain classes of form constraints. They provide a way of configuring a component so that it spans either the full height or width of the form widget. This behavior is accomplished by attaching two edges of the component (top and bottom for `DtSpanHeight`, and left and right for `DtSpanWidth`) to the form widget. The component typically resizes whenever the form widget is resized. The `ATTACH_FORM` constraint is used for all attachments.

Usage:

```
DtkshSpanWidth [leftOffset rightOffset]
```

```
DtkshSpanHeight [topOffset bottomOffset]
```

The optional *offset* parameters specify how far from the edges of the form widget the component should be positioned. If an offset is not specified, then 0 is used.

Example:

```
XtCreateManagedWidget SEP sep XmSeparator $FORM \
                      $(DtkshSpanWidth 1 1)
```

DtkshDisplayInformationDialog, DtkshDisplayQuestionDialog, DtDisplayWarningDialog, DtkshDisplayWorkingDialog, and DtkshDisplayErrorDialog

These convenience functions create a single instance of each of the Motif feedback dialogs. If an instance of the requested type of dialog already exists, then it is reused. The parent of the dialog is obtained from the environment variable `$TOPLEVEL`, which should be set by the calling shell script, and then should not be changed. The handle for the requested dialog is returned in one of the following environment variables:

- `_DTKSH_ERROR_DIALOG_HANDLE`
- `_DTKSH_QUESTION_DIALOG_HANDLE`
- `_DTKSH_WORKING_DIALOG_HANDLE`
- `_DTKSH_WARNING_DIALOG_HANDLE`
- `_DTKSH_INFORMATION_DIALOG_HANDLE`

Note – If you are attaching your own callbacks to the dialog buttons, do not destroy the dialog when you are done with it. Unmanage the dialog, so that it can be used again at a later time. If it is necessary to destroy the dialog, then be sure to clear the associated environment variable so the convenience function does not attempt to reuse the dialog.

Usage:

```
DtkshDisplay<name>Dialog title message [okCallback closeCallback
                        helpCallback dialogStyle]
```

The Ok button is always managed, and by default unmanages the dialog. The Cancel and Help buttons are only managed when a callback is supplied for them. The *dialogStyle* parameter accepts any of the standard resource settings supported by the associated bulletin board resource.

Example:

```
DtkshDisplayErrorDialog "Read Error" "Unable to read the file"
                 "OkCallback" \
               "CancelCallback" "" DIALOG_PRIMARY_APPLICATION_MODAL
```

DtkshDisplayQuickHelpDialog and DtkshDisplayHelpDialog

These convenience functions create a single instance of each of the help dialogs. If an instance of the requested type of help dialog already exists, then it is reused. The parent of the dialog is obtained from the environment variable `$TOPLEVEL`, which should be set by the calling shell script, and then should not be changed. The handle for the requested dialog is returned in one of the following environment variables:

- `_DTKSH_HELP_DIALOG_HANDLE`
- `_DTKSH_QUICK_HELP_DIALOG_HANDLE`

Note – If it is necessary to destroy a help dialog, then be sure to clear the associated environment variable so that the convenience function does not attempt to reuse the dialog.

Usage:

```
DtkshDisplay*HelpDialog title helpType helpInformation [locationId]
```

The meaning of the parameters is dependent upon the value specified for the *helpType* parameter. Their meanings are:

- *helpType* = `HELP_TYPE_TOPIC`
  - *helpInformation* = help volume name
  - *locationId* = help topic location ID
- *helpType* = `HELP_TYPE_STRING`
  - *helpInformation* = help string
  - *locationId* = <not used>
- *helpType* = `HELP_TYPE_DYNAMIC_STRING`
  - *helpInformation* = help string
  - *locationId* = <not used>
- *helpType* = `HELP_TYPE_MAN_PAGE`
  - *helpInformation* = manual page name
  - *locationId* = <not used>
- *helpType* = `HELP_TYPE_FILE`
  - *helpInformation* = help file name
  - *locationId* = <not used>

Example:

```
DtkshDisplayHelpDialog "Help On Dtksh" HELP_TYPE_FILE
                        "helpFileName"
```

The script_find Script

C

This appendix contains the complete listing of `script_find` described in Chapter 4, "A Complex Script." The script executes a second script called `Find.sticky`, which is listed after `script_find`. There is also a file called `Find.help`, which is a text file accessed when the user clicks the Help button on the main script window. See Chapter 4 for more information on this script.

Listing for script_find

```
#! /usr/dt/bin/dtksh
set -u

. /usr/dt/lib/dtksh/DtFuncs.dtsh

#
# This sample shell script provides a graphical interface to the
# 'find' command.  Each time it is executed, it will attempt to
# restore the dialog to the last set of values entered by the user.
# When the 'find' command is initiated, the output will be displayed
# in a dtterm window.
#

#
# Post an# error dialog.  The main application window is disabled
# until the error dialog is unposted.  The message to be displayed
# in the # error dialog is passed in as $1
#
PostErrorDialog()
{
```

```
      DtDisplayErrorDialog "Find Error" "$1" \
      DIALOG_PRIMARY_APPLICATION_MODAL
}

#
# This is both the 'Ok' and the 'Apply' callback; in the case of the
# 'Ok' callback, it  unposts the main application window, and then
# exits, if the dialog contains valid information.  For both 'Ok' and
# 'Apply', the set of search directories is first validated; if any
# of the paths are not valid, then an error dialog is posted.
# Otherwise, the 'find' process is started in a terminal window.
#
OkCallback()
{
   RetrieveAndSaveCurrentValues
   if [ "$SD_VAL" = "" ] ; then
      PostErrorDialog "You must specify a directory to search"
   else
      for i in $SD_VAL ; do
         if  [ ! -d $i ] ; then
            MSG="The following search directory does not exist:

            $i"
            PostErrorDialog "$MSG"
            return 1
         fi
      done

      if [ $CB_WIDGET = $OK ] ; then
         XtPopdown $TOPLEVEL
      fi

      CMD="/bin/find $SD_VAL"
      if [ ! "$FNP_VAL" = "" ] ; then
         CMD=$CMD" -name $FNP_VAL"
      fi

      if ! $(XmToggleButtonGetState $T1); then
         CMD=$CMD" -xdev"
      fi

      if $(XmToggleButtonGetState $T3); then
         CMD=$CMD" -hidden"
      fi

      if $(XmToggleButtonGetState $T4); then
```

```
            CMD=$CMD" -follow"
         fi

         if $(XmToggleButtonGetState $T5); then
            CMD=$CMD" -depth"
         fi

         case $FSTYPE_VAL in
            $NFS)  CMD=$CMD" -fsonly nfs" ;;
            $CDFS) CMD=$CMD" -fsonly cdfs" ;;
            $HFS)  CMD=$CMD" -fsonly hfs" ;;
            *) ;;
         esac

         case $FILETYPE_VAL in
            $REGULAR)   CMD=$CMD" -type f" ;;
            $DIRECTORY) CMD=$CMD" -type d" ;;
            $BLOCK)     CMD=$CMD" -type b" ;;
            $CHAR)      CMD=$CMD" -type c" ;;
            $FIFO)      CMD=$CMD" -type p" ;;
            $SYMLINK)   CMD=$CMD" -type l" ;;
            $SOCKET)    CMD=$CMD" -type s" ;;
            $NET)       CMD=$CMD" -type n" ;;
            $MOUNT)     CMD=$CMD" -type M" ;;
            $HIDDEN)    CMD=$CMD" -type H" ;;
            *) ;;
         esac

         if $(XmToggleButtonGetState $T2); then
            CMD=$CMD" -print"
         fi

         /usr/dt/bin/dtterm -title "Find A File"
           -e /usr/dt/bin/dtexec -open -1 $CMD &

         if [ $CB_WIDGET = $OK ] ; then
            exit 0
         fi
      fi
}


#
# This function attempt to load in the previous dialog values.
# Each line read from the file is then interpreted as a ksh command.
#
```

```
LoadStickyValues()
{
   if [ -r "./Find.sticky" ] ; then
      exec 6< "./Find.sticky"
      XtAddInput FID 6 "EvalCmd"
   fi
}


#
# This function is invoked for each line in the 'sticky' values file.
# It will evaluate each line as a dtksh command.
#
EvalCmd()
{
   if [ ${#INPUT_LINE} -gt 0 ]; then
      eval "$INPUT_LINE"
   fi

   if [ "$INPUT_EOF" = 'true' ]; then
      XtRemoveInput $INPUT_ID
      eval exec $INPUT_SOURCE'<&-'
   fi
}


#
# This function retrieves the current values, and then saves them
# off into a file, so that they can be restored the next time the
# dialog is displayed.  It is called anytime the user selects either
# the "Ok" or "Apply" buttons.
#
RetrieveAndSaveCurrentValues()
{
   XmTextGetString SD_VAL $SD
   XmTextGetString FNP_VAL $FNP
   XtGetValues $FSTYPE menuHistory:FSTYPE_VAL
   XtGetValues $FILETYPE menuHistory:FILETYPE_VAL

   exec 3> "./Find.sticky"
   if [ ! "$SD_VAL" = "" ] ; then
      print -u 3 "XmTextSetString \$SD \"$SD_VAL\""
      print -u 3 "XmTextFieldSetInsertionPosition \$SD ${#SD_VAL}"
   fi
   if [ ! "$FNP_VAL" = "" ] ; then
      print -u 3 "XmTextSetString \$FNP \"$FNP_VAL\""
```

```
  print -u 3 "XmTextFieldSetInsertionPosition \$FNP ${#FNP_VAL}"
fi

case $FSTYPE_VAL in
   $NFS)  FST="\$NFS" ;;
   $CDFS) FST="\$CDFS" ;;
   $HFS)  FST="\$HFS" ;;
   *)     FST="\$NODIR" ;;
esac
print -u 3 "XtSetValues \$FSTYPE menuHistory:$FST"

case $FILETYPE_VAL in
   $REGULAR)   FT="\$REGULAR" ;;
   $DIRECTORY) FT="\$DIRECTORY" ;;
   $BLOCK)     FT="\$BLOCK" ;;
   $CHAR)      FT="\$CHAR" ;;
   $FIFO)      FT="\$FIFO" ;;
   $SYMLINK)   FT="\$SYMLINK" ;;
   $SOCKET)    FT="\$SOCKET" ;;
   $NET)       FT="\$NET" ;;
   $MOUNT)     FT="\$MOUNT" ;;
   $HIDDEN)    FT="\$HIDDEN" ;;
   *)          FT="\$NOTYPE" ;;
esac
print -u 3 "XtSetValues \$FILETYPE menuHistory:$FT"

if $(XmToggleButtonGetState $T1); then
   print -u 3 "XmToggleButtonSetState \$T1 true false"
fi

if $(XmToggleButtonGetState $T2); then
   print -u 3 "XmToggleButtonSetState \$T2 true false"
fi

if $(XmToggleButtonGetState $T3); then
   print -u 3 "XmToggleButtonSetState \$T3 true false"
fi

if $(XmToggleButtonGetState $T4); then
   print -u 3 "XmToggleButtonSetState \$T4 true false"
fi

if $(XmToggleButtonGetState $T5); then
   print -u 3 "XmToggleButtonSetState \$T5 true false"
fi
```

```
   exec 3<&-
}


################ Create the Main UI ####################


set -f
XtInitialize TOPLEVEL find Dtksh $0 "${@:-}"
XtSetValues $TOPLEVEL title:"Find Files"

XtCreateManagedWidget FORM form XmForm $TOPLEVEL

XtCreateManagedWidget SDLABEL sdlabel XmLabel $FORM \
   labelString:"Search Directory:" \
   $(DtkshAnchorTop 12) \
   $(DtkshAnchorLeft 10)

XtCreateManagedWidget SD sd XmText $FORM \
   columns:30 \
   value:"." \
   $(DtkshAnchorTop 6) \
   $(DtkshRightOf $SDLABEL 10) \
   $(DtkshAnchorRight 10) \
   navigationType:EXCLUSIVE_TAB_GROUP
XmTextFieldSetInsertionPosition $SD 1

XtCreateManagedWidget FNPLABEL fnpabel XmLabel $FORM \
   labelString:"Filename Pattern:" \
   $(DtkshUnder $SDLABEL 24) \
   $(DtkshAnchorLeft 10)

XtCreateManagedWidget FNP fnp XmText $FORM \
   columns:30 \
   $(DtkshUnder $SD 8) \
   $(DtkshRightOf $FNPLABEL 10) \
   $(DtkshAnchorRight 10) \
   navigationType:EXCLUSIVE_TAB_GROUP

XtCreateManagedWidget SEP sep XmSeparator $FORM \
   separatorType:SINGLE_DASHED_LINE \
   $(DtkshUnder $FNP 10) \
   $(DtkshSpanWidth)

XtCreateManagedWidget RC rc XmRowColumn $FORM \
        orientation:HORIZONTAL \
```

```
        numColumns:3 \
        packing:PACK_COLUMN \
   $(DtkshUnder $SEP 10) \
   $(DtkshSpanWidth 10 10) \
   navigationType:EXCLUSIVE_TAB_GROUP

DtkshAddButtons -w $RC XmToggleButtonGadget \
   T1 “Cross Mount Points”            ""\
   T2 “Print Matching Filenames”      ""\
   T3 “Search Hidden Subdirectories” ""\
   T4 “Follow Symbolic Links”         ""\
   T5 “Descend Subdirectories First” ""

XtCreateManagedWidget SEP2 sep XmSeparator $FORM \
   separatorType:SINGLE_DASHED_LINE \
   $(DtkshUnder $RC 10) \
   $(DtkshSpanWidth)

XmCreatePulldownMenu PANE $FORM pane
DtkshAddButtons -w $PANE XmPushButtonGadget \
   NODIR “no restrictions” ""\
   NFS   “nfs”             ""\
   CDFS  “cdfs”            ""\
   HFS   “hfs”             ""

XmCreateOptionMenu FSTYPE $FORM fstype \
        labelString:”Restrict Search To File System Type:” \
        menuHistory:$NODIR \
        subMenuId:$PANE \
   $(DtkshUnder $SEP2 20) \
   $(DtkshSpanWidth 10 10) \
   navigationType:EXCLUSIVE_TAB_GROUP
XtManageChild $FSTYPE

XmCreatePulldownMenu PANE2 $FORM pane2
DtkshAddButtons -w $PANE2 XmPushButtonGadget \
   NOTYPE    “no restrictions”   ""\
   REGULAR   “regular”           ""\
   DIRECTORY “directory”         ""\
   BLOCK     “block special”     ""\
   CHAR      “character special” ""\
   FIFO      “fifo”              ""\
   SYMLINK   “symbolic link”     ""\
   SOCKET    “socket”            ""\
   NET       “network special”   ""\
   MOUNT     “mount point”       ""\
```

```
   HIDDEN    “hidden directory”  ““

XmCreateOptionMenu FILETYPE $FORM filetype \
        labelString:”Match Only Files Of Type:” \
        menuHistory:$NOTYPE \
        subMenuId:$PANE2 \
   $(DtkshUnder $FSTYPE 10) \
   $(DtkshSpanWidth 10 10) \
   navigationType:EXCLUSIVE_TAB_GROUP
XtManageChild $FILETYPE
XtSetValues $FILETYPE spacing:90

XtCreateManagedWidget SEP3 sep3 XmSeparator $FORM \
   $(DtkshUnder $FILETYPE 10) \
   $(DtkshSpanWidth)

XtCreateManagedWidget OK ok XmPushButton $FORM \
        labelString:”Ok” \
   $(DtkshUnder $SEP3 10) \
   $(DtkshFloatLeft 4) \
   $(DtkshFloatRight 24) \
   $(DtkshAnchorBottom 10)
XtAddCallback $OK activateCallback “OkCallback”

XtCreateManagedWidget APPLY apply XmPushButton $FORM \
        labelString:”Apply” \
   $(DtkshUnder $SEP3 10) \
   $(DtkshFloatLeft 28) \
   $(DtkshFloatRight 48) \
   $(DtkshAnchorBottom 10)
XtAddCallback $APPLY activateCallback “OkCallback”

XtCreateManagedWidget CLOSE close XmPushButton $FORM \
        labelString:”Close” \
   $(DtkshUnder $SEP3 10) \
   $(DtkshFloatLeft 52) \
   $(DtkshFloatRight 72) \
   $(DtkshAnchorBottom 10)
XtAddCallback $CLOSE activateCallback “exit 1”

XtCreateManagedWidget HELP help XmPushButton $FORM \
        labelString:”Help” \
   $(DtkshUnder $SEP3 10) \
   $(DtFloatLeft 76) \
   $(DtkshFloatRight 96) \
   $(DtkshAnchorBottom 10)
```

```
XtAddCallback $HELP activateCallback \
       "DtkshDisplayQuickHelpDialog 'Using The Find Command'
HELP_TYPE_FILE \
   './Find.help' "

XtSetValues $FORM \
   initialFocus:$SD \
   defaultButton:$OK \
   cancelButton:$CLOSE \
   navigationType:EXCLUSIVE_TAB_GROUP

DtkshSetReturnKeyControls $SD $FNP $FORM $OK
LoadStickyValues

XtRealizeWidget $TOPLEVEL
XtMainLoop
```

Find.sticky

The following script, `Find.sticky` is executed by `script_find`. `Find.sticky` remembers the file and directory names used in the most recent execution of `script_find`.

```
XmTextSetString $SD "/users/dlm"
XmTextFieldSetInsertionPosition $SD 10
XmTextSetString $FNP "elmbug"
XmTextFieldSetInsertionPosition $FNP 6
XtSetValues $FSTYPE menuHistory:$NODIR
XtSetValues $FILETYPE menuHistory:$DIRECTORY
XmToggleButtonSetState $T1 true false
XmToggleButtonSetState $T2 true false
```

Find.help

`Find.help` is a text file that is displayed on screen when the user clicks the Help button in the main `script_find` window.

```
This dialog presents a graphical interface to the
UNIX 'find' command.  The only required field is
the name of the directory to be searched;
all other fields are optional.  Once the fields have
been set to the desired values, you can use the
'Ok' or 'Apply' button to initiate the find operation.
The results of the find operation are displayed
in a dtterm terminal window.
```

Index