VMS DECwindows User Interface Language Reference Manual

Order Number: AA–MG22B–TE

June 1990

This manual describes the syntax and usage of language elements and module components of the User Interface Language (UIL). It also describes how to use the UIL compiler and how to interpret compilation diagnostics. This manual contains a listing of the UIL built-in tables used during compilation to check that your UIL specification is consistent with the XUI Toolkit.

Revision/Update Information:	This manual supersedes the *VMS DECwindows User Interface Language Reference Manual*, Version 5.3.
Software Version:	VMS Version 5.4

digital equipment corporation
maynard, massachusetts

June 1990

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CDA	DEQNA	MicroVAX	VAX RMS
DDIF	Desktop–VMS	PrintServer 40	VAXserver
DEC	DIGITAL	Q-bus	VAXstation
DECdtm	GIGI	ReGIS	VMS
DECnet	HSC	ULTRIX	VT
DECUS	LiveLink	UNIBUS	XUI
DECwindows	LN03	VAX	
DECwriter	MASSBUS	VAXcluster	digital™

The following are third-party trademarks:

PostScript is a registered trademark of Adobe Systems Incorporated.

X Window System, Version 11 and its derivations (X, X11, X Version 11, X Window System) are trademarks of the Massachusetts Institute of Technology.

ZK5001

Production Note

This book was produced with the VAX DOCUMENT electronic publishing system, a software tool developed and sold by Digital. In this system, writers use an ASCII text editor to create source files containing text and English-like code; this code labels the structural elements of the document, such as chapters, paragraphs, and tables. The VAX DOCUMENT software, which runs on the VMS operating system, interprets the code to format the text, generate a table of contents and index, and paginate the entire document. Writers can print the document on the terminal or line printer, or they can use Digital-supported devices, such as the LN03 laser printer and PostScript printers (PrintServer 40 or LN03R ScriptPrinter), to produce a typeset-quality copy containing integrated graphics.

Contents

TABLES

Preface

This manual describes the syntax and features of the User Interface Language (UIL), a specification language for describing the initial state of a user interface for a VMS DECwindows application.

Intended Audience

This manual is intended for application programmers who are familiar with the routines in the XUI Toolkit and who know at least one high-level programming language.

Document Structure

This manual includes four chapters and two appendixes.

- Chapter 1 introduces UIL and explains the advantages UIL offers over other methods for creating user interfaces for VMS DECwindows applications.
- Chapter 2 describes the syntax of low-level elements of UIL, including names, keywords, literals, functions, supported character sets for string literals, and compile-time value expressions.
- Chapter 3 describes the syntax of UIL module components (module header, value, procedure, identifier, and object declarations; arguments, controls, and callbacks lists) and describes additional features of UIL, including support for constraint widgets, symbolic references to widget identifiers, and UIL identifiers.
- Chapter 4 explains how to use the UIL compiler, including UIL command line qualifiers, and how to interpret compiler diagnostics.
- Appendix A lists diagnostic messages and includes corrective action.
- Appendix B lists the UIL built-in tables. For each object in the XUI Toolkit, this appendix lists the callback reasons, children, and arguments supported by the UIL compiler. The data type and default value are given for each argument. This appendix also includes an alphabetical listing of the arguments with their MIT C and VAX binding names.

Associated Documents

The following manuals contain information on the XUI Resource Manager (DRM) run-time routines (which read the output of the UIL compiler) and on using UIL and DRM together to create the interface for a DECwindows application.

- *VMS DECwindows Guide to Application Programming*
- *VMS DECwindows Toolkit Routines Reference Manual*

The following documents include information that may be helpful in using UIL to specify a user interface.

- *XUI Style Guide*
- *VMS DECwindows Xlib Programming Volume*
- *VMS DECwindows Xlib Routines Reference Manual*

Conventions

The following conventions are used in this manual:

mouse	The term *mouse* is used to refer to any pointing device, such as a mouse, a puck, or a stylus.
Ctrl/x	A sequence such as Ctrl/x indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
Return	In examples, a key name is shown enclosed in a box to indicate that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	In examples, a horizontal ellipsis indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
.	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[ ]	In format descriptions, brackets indicate that whatever is enclosed within the brackets is optional; you can select none, one, or all of the choices. (Brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
{ }	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
red ink	Red ink indicates information that you must enter from the keyboard or a screen object that you must choose or click on. For online versions of the book, user input is shown in **bold**.

boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason. Boldface text is also used to show user input in online versions of the book.
italic text	Italic text represents information that can vary in system messages (for example, Internal error *number*).
UPPERCASE TEXT	Uppercase letters indicate that you must enter a command (for example, enter OPEN/READ), or they indicate the name of a routine, the name of a file, the name of a file protection code, or the abbreviation for a system privilege.
-	Hyphens in coding examples indicate that additional arguments to the request are provided on the line that follows.
numbers	Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

Syntax of the UIL language elements and the UIL module components in this manual are described using the following extended Backus-Naur Form (BNF) notation:

Notation	Description
VALUE	Keywords are shown in uppercase letters.
k_main_menu	Lowercase words represent either user-supplied names and constants or complete syntactical entries defined elsewhere in the syntax.
[MANAGED | UNMANAGED]	Brackets enclose an optional part of the syntax.
{widget_name}	Braces enclose a required part of the syntax.
EXPORTED | PRIVATE | IMPORTED	A vertical bar indicates mutually exclusive alternatives.
list_declaration...	Ellipsis (...) indicates that the clause can be repeated either zero or more times if it is optional, or one or more times if it is required.
"="	Punctuation that is part of the syntax is enclosed in quotation marks.
xxx_name	A name in this form must correspond to a valid UIL name.

1 Introduction to the User Interface Language

This chapter introduces the User Interface Language (UIL) and explains the advantages UIL offers over using XUI Toolkit routines to create user interfaces for VMS DECwindows applications.

1.1 Overview of UIL

UIL is a specification language for describing the initial state of a user interface for a DECwindows application. The specification describes the objects (menus, dialogs, labels, push buttons, and so on) used in the interface and specifies the routines to be called when the interface changes state as a result of user interaction.

The UIL compiler translates the UIL code into a User Interface Definition (UID) file. You include XUI Resource Manager (DRM) routine calls in your application program, which allow access to the UID file. During execution of the application, DRM builds argument lists and makes the necessary calls to the widget creation routines in order to create the user interface. UIL and DRM are components of the XUI Toolkit.

Using UIL, you can specify the following:

- Objects (widgets and gadgets) that comprise your interface
- Arguments (attributes) of the widgets and gadgets you specify
- Callback routines for each widget
- Hierarchy of objects in your application
- Literal values that can be fetched by the application

The UIL compiler has built-in tables containing information about widgets and gadgets. For every object in the XUI Toolkit, the UIL compiler knows which objects are valid children of the object, the object's arguments, and the valid callback reasons for the object.

1.2 Advantages to Using UIL

Creating a user interface for a VMS DECwindows application using UIL offers the following advantages:

Easier Coding

You can specify an interface more quickly using UIL because you do not have to know the specific creation routines or the format of their argument lists. You need to include only those object arguments you want to change. In general, you can specify these arguments in any order.

1.2 Advantages to Using UIL

Because UIL is a specification language that describes the characteristics of an interface, there is no need for control flow. Therefore, you can define objects in your UIL specification in roughly the same order that the objects are arranged in the widget hierarchy of your application interface. This makes it easier for a programmer reading the UIL specification to interpret the design of the interface.

At run time, when the interface objects are created, DRM performs some Toolkit routine calls for you as a convenience, thus simplifying your programming tasks.

Earlier Error Detection

The UIL compiler does type checking for you that is not available with the XUI or X Toolkits, so that the interface you specify has fewer errors.

The UIL compiler issues diagnostics if you specify any of the following:

- The wrong type of value for an argument
- An argument to an object that is not supported by that object
- A reason for an object that the object does not support
- A child of an object that the object does not support

Separation of Form and Function

When you use UIL, you define your application interface in a separate UIL module rather than by directly calling XUI Toolkit creation routines in your application program. This lets you separate the form your interface takes from the functions provided by the application. By separating form and function, it becomes feasible to have multiple interfaces that share a common set of functions. This is beneficial, for example, when you are building an application that will be used by people who speak different languages.

In general, you can freely change the appearance of the interface (for example, by repositioning widgets or changing their borders or colors) without recompiling the application program.

Faster Prototype Development

UIL helps you develop prototypes of user interfaces for an application. You can create a variety of interfaces in a fairly short time. You can get an idea of the look of each interface before any of the functional routines are written.

The ability to specify the user interface separately lets designers work with end users at the same time programmers are coding the functions of the application. Because both groups can work more or less independently, the complete application can be delivered in less time than if the interface design were part of the application code.

Interface Customization

You can customize an interface by putting in place a hierarchy of UID files (called a UID hierarchy). At run time, DRM searches this file hierarchy in the sequence you specify to build the appropriate argument lists for the low-level widget creation routines.

One use of this feature would be to provide an interface in several languages. The text on title bars, menus, and so on, can be displayed in the language of the end user without altering anything in the application. In this case, the files in the UID hierarchy represent alternative interfaces.

Another use of the UID hierarchy feature would be to isolate individual, department, and division customizations to an interface. In this case, you can think of the files in the UID hierarchy as superimposed, with the definitions in the first file listed in the array supplied to the DRM routine OPEN HIERARCHY taking precedence.

1.3 Features of UIL

UIL offers many features to increase productivity and the flexibility of your programs.

Named Values

Instead of directly specifying the values for widget and gadget attributes, you can use named values, which are similar to variables in a programming language. You give a literal value (such as an integer or string) a name and then use the name in place of the value specification. Using named values makes your UIL specification easier to understand and isolates changes.

In addition, you can use DRM routines to fetch named values from the UID file for use at run time.

Compile-Time Expressions

You can use expressions to specify values in UIL. A valid UIL expression can contain integers, strings, floating-point numbers, Booleans, named values, and operators. Using expressions can make values more descriptive (for example, "*dialog_box_width*/2") and can help you avoid recomputing values (for example, if you needed to change the size or position of the dialog box).

Identifiers

Identifiers provide a mechanism for referencing values in UIL which are provided by the application at run time. In the application program, you can use a DRM routine to associate a value with the identifier name. Unlike a named value, an identifier does not have an associated data type. You can use an identifier as an attribute value or callback procedure tag, regardless of the data type specified in the object or procedure declaration. Identifiers are useful for specifying position based upon the type of terminal on which the interface will be displayed or for passing a data structure (as opposed to a constant) to a callback procedure.

Lists

UIL allows you to create named lists of attributes, sibling widgets, and callback procedures that you can later refer to by name. This feature allows you to easily reuse common definitions by simply referencing these definitions by name.

Support for Compound Strings

Most XUI Toolkit widgets require strings used in the user interface (labels, menu items, and so on) to be compound strings. UIL fully supports use of compound strings, including left-to-right and right-to-left writing direction and choice of fonts.

Include Files for Useful Constants

The XUI Toolkit provides include files (one for each of the VMS and MIT C bindings) that contain useful constants for coding a user interface in UIL. For example, several widgets have an orientation attribute. You can use the constants **DwtOrientationHorizontal** or **DwtOrientationVertical** (in the MIT C binding) to specify the orientation attributes.

1.4 Where to Find More Information

This manual contains reference information on the User Interface Language and the UIL compiler. For information on specifying an interface using UIL and creating the interface at run time using DRM, see the *VMS DECwindows Guide to Application Programming*.

For reference information on DRM routines, see the *VMS DECwindows Toolkit Routines Reference Manual*.

2 UIL Language Syntax

This chapter describes the syntax rules for the following low-level language elements:

- Character set
- Names
- Keywords
- Literals
- Value-generating functions
- ANY data type

The chapter also describes include files for UIL constants, character sets for compound strings, and compile-time value expressions.

UIL is a free-form language. This means that high-level constructs such as object and value declarations do not need to begin in any particular column and can span any number of lines. Low-level constructs such as keywords and punctuation characters can also begin in any column; however, except for string literals and comments, they cannot span lines.

The UIL compiler accepts input lines up to 132 characters in length.

2.1 Character Set and Punctuation Characters

Elements of UIL are constructed using the characters in Table 2–1.

Table 2–1 UIL Character Set

Alphabetic characters	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Numeric characters	0123456789
UIL punctuation and formatting characters	_' ( )*,+ –./;:=!$\{} space, tab, form feed
Other characters	&[] | <>%"#?@~'^ DEC Multinational Character in the range 160 to 255 decimal

Alphabetic, numeric, and punctuation and formatting characters are used to build the elements of UIL. Other characters are valid only in comments and string literals.

Note: Control characters (except for the form-feed character) are not permitted in a UIL module; the form-feed character is permitted in column 1 only. Escape sequences (described in Section 2.4) must be used to construct a string literal containing a control character.

2.1.1 Punctuation Characters

Use the character sequences shown in Table 2–2 to punctuate a UIL module.

Table 2–2 UIL Punctuation Characters

(	Left parenthesis	)	Right parenthesis
{	Left brace	}	Right brace
\	Backslash	!	Exclamation mark
/*	Slash asterisk	*/	Asterisk slash
;	Semicolon	:	Colon
'	Apostrophe	,	Comma
=	Equal sign		

Punctuation in a UIL module resembles that used in C programs. For example, statements end in a semicolon (;), braces ({}) are used to delimit definitions, and comments can be delimited by the /* and */ character sequences.

2.1.2 Spaces, Tabs, Form-Feed Characters, and Comments

Spaces, tabs, and comments are special elements in the language. They are a means of delimiting other elements, such as two names. One or more of these elements may appear before or after any other element in the language. However, spaces, tabs, and comments that appear in string literals are treated as character sequences rather than as delimiters.

Comments can take one of two forms, as follows:

- The comment is introduced with the sequence /* followed by the text of the comment and terminated with the sequence */. This form of comment may span multiple source lines.
- The comment is introduced with the character ! followed by the text of the comment and terminated by the end of the source line.

Neither form of comment can be nested.

The form-feed character is a control character and therefore cannot appear directly in a UIL specification file. You must use the escape sequence \12\ instead (see Table 2–6). There is one exception to this rule; you can place a form feed character in column 1 of a source line due to the common practice of VMS editors separating parts of a program with a form feed. The form feed causes a page break in the module listing.

2.2 Names

Each entity in UIL (such as a value, procedure, or object) can be identified by a name. This name is also used to reference the entity elsewhere in the UIL module.

Names can consist of any of the characters A through Z, a through z, 0 through 9, dollar sign ($), and underscore (_). Names cannot begin with a digit (0 through 9). The maximum length of a name is 31 characters.

UIL gives you a choice of either case-sensitive or case-insensitive names through a clause in the module header (described in Section 3.1.2). For example, if names are case sensitive, then the names "sample" and "Sample" are distinct from each other. If names are case insensitive, these names would be treated as the same name and could be used interchangeably.

In case-insensitive mode, the compiler outputs all names in the User Interface Definition (UID) file in uppercase form.

In case-sensitive mode, two names are the same only if the names contain exactly the same characters. In this mode, names appear in the UID file exactly as they appear in the source.

By default, names are case insensitive.

UIL has a single name space; therefore, you must define any referenced name exactly once in a UIL module. If you inadvertently define the same name more than once or omit a definition, the UIL compiler issues an error at compilation time.

2.3 Keywords

Keywords are a set of names that have special meaning in UIL. They are of two types: reserved and nonreserved. You cannot use a reserved keyword to name an entity in UIL. Nonreserved keywords can be used as names.

If you specify case-insensitive mode, you can type UIL keywords in either uppercase, lowercase, or mixed case.

If you specify case-sensitive mode, you must type UIL keywords in lowercase.

Keywords cannot be abbreviated by truncating characters from the end.

Table 2–3 lists reserved keywords in alphabetical order.

Table 2–3 Reserved UIL Keywords

Keyword	Description
ARGUMENTS	Identifies an arguments list.
CALLBACKS	Identifies a callbacks list.

(continued on next page)

Table 2–3 (Cont.) Reserved UIL Keywords

Keyword	Description
CONTROLS	Identifies a controls list.
END	Signifies the end of the module.
EXPORTED	Specifies that this object or value can be referenced by other UIL modules.
FALSE	Represents the Boolean value 0; synonym for OFF.
GADGET	For objects having a widget and a gadget variant, specifies this object as the gadget variant.
IDENTIFIER	Indicates an identifier declaration.
INCLUDE	Used with the nonreserved keyword FILE to specify an include file.
LIST	Identifies a list declaration.
MODULE	Signifies the start of a UIL module.
OBJECT	Identifies an object declaration.
OFF	Represents the Boolean value 0; synonym for FALSE.
ON	Represents the Boolean value 1; synonym for TRUE.
PRIVATE	Specifies that this object or value cannot be referenced by other UIL modules.
PROCEDURE	Identifies a procedure declaration.
PROCEDURES	Identifies a procedures list.
TRUE	Represents the Boolean value 1; synonym for ON.
VALUE	Identifies a value (literal) declaration.
WIDGET	For objects having a widget and a gadget variant, specifies this object as the widget variant.

Keywords listed in Table 2–4 are nonreserved keywords that you can use as names without generating an error. Note, however, that if you use any of these keywords as a name, you cannot use the UIL-supplied meaning of that keyword. For example, if you use the name of an argument (such as **x**) as the name of a value, you cannot specify the **x** argument in any object definitions.

Table 2–4 Nonreserved UIL Keywords

Keyword	Description
Built-in argument names[1] (for example: **x**, **height**)	Identifies an object argument (widget-specific attribute).
Built-in reason names[1] (for example: **activate**, **help**)	Identifies a callback reason.

[1]Appendix B lists built-in argument names, reason names, and object types.

(continued on next page)

Table 2–4 (Cont.) Nonreserved UIL Keywords

Keyword	Description
Character set names[2] (for example: ISO_LATIN1)	Identifies a character set and its writing direction.
Object types[1] (for example: push_button, dialog_box)	Identifies a DECwindows interface object.
ANY	Suppresses data type checking.
ARGUMENT	Identifies the built-in ARGUMENT function.
ASCIZ_STRING_TABLE	Specifies a value as the UIL data type **asciz_table**.
ASCIZ_TABLE	Specifies a value as the UIL data type **asciz_table**.
BACKGROUND	In a color table, specifies monochrome mapping to the background color.
BOOLEAN	Specifies a literal as UIL data type **boolean**.
CASE_INSENSITIVE	Used with the nonreserved keyword NAMES to specify that names and keywords in the module are case insensitive.
CASE_SENSITIVE	Used with the nonreserved keyword NAMES to specify that names and keywords in the module are case sensitive.
CLASS_REC_NAME	Specifies a value as the UIL data type **class_rec_name**.
COLOR	Specifies a value as the UIL data type **color**.
COLOR_TABLE	Specifies a value as the UIL data type **color_table**.
COMPOUND_STRING	Specifies a value as the UIL data type **compound_string**; identifies the built-in COMPOUND_STRING data conversion function.
COMPOUND_STRING_TABLE	Specifies a value as the UIL data type **string_table**.
FILE	Used with the reserved keyword INCLUDE to specify an include file.
FLOAT	Specifies a literal as UIL data type **float**; identifies the FLOAT data conversion function.
FONT	Specifies a value as the UIL data type **font**.
FONT_TABLE	Specifies a value as the UIL data type **font_table**.
FOREGROUND	In a color table, specifies monochrome mapping to the foreground color.
ICON	Specifies a value as the UIL data type **pixmap**; identifies the built-in ICON function.
IMPORTED	Specifies that this literal takes its value from a corresponding literal in another UIL module.

[1]Appendix B lists built-in argument names, reason names, and object types.

[2]Table 2–7 lists UIL character set names.

(continued on next page)

Table 2–4 (Cont.) Nonreserved UIL Keywords

Keyword	Description
INTEGER	Specifies a literal as UIL data type **integer**; identifies the INTEGER data conversion function.
INTEGER_TABLE	Specifies a value as the UIL data type **integer_table**.
MANAGED	Specifies that a child is managed by its parent.
NAMES	Identifies the case-sensitivity clause.
OBJECTS	Identifies the default object variant clause.
PIXMAP	Specifies a value as the UIL data type **pixmap**.
REASON	Specifies a value as the UIL data type **reason**.
RIGHT_TO_LEFT	Specifies writing direction in the COMPOUND_STRING function.
STRING	Specifies a literal as the UIL data type **string**.
STRING_TABLE	Specifies a value as the UIL data type **string_table**.
TRANSLATION_TABLE	Specifies a value as the UIL data type **translation_table**.
UNMANAGED	Specifies that a child is unmanaged by its parent.
VERSION	Identifies the version clause.
XBITMAPFILE	Specifies a value as the UIL data type **pixmap**; identifies the XBITMAPFILE function.

Note: In this chapter, all examples assume case-insensitive mode. Keywords are shown in uppercase letters to distinguish them from user-specified names, which are shown in lowercase letters. This use of uppercase letters is not required in case-insensitive mode. In case-sensitive mode, keywords *must* be in *lowercase* letters.

In the following example, ARGUMENTS is a keyword and *circle_radius* is a user-defined name having the integer value 1000. In your UIL module, you could type the keyword ARGUMENTS in lowercase, uppercase, or mixed-case letters as long as you specified that names are case insensitive.

```
    .
    .
    .
{ ARGUMENTS
    { circle_radius = 1000; };
    .
    .
    .
```

2.4 Literals

Literals are one means of specifying a value. UIL provides literals for several of the value types it supports. Some of the value types are not supported as literals (for example, pixmaps and string tables). You can specify values for these types by using functions provided by UIL (described in Section 2.5). Literal types supported by UIL are as follows:

- String
- Integer
- Boolean
- Floating-point

You can designate UIL objects and values as exported, imported, or private. An exported object or value can be referenced in another UIL module by using the same name for the object or value and indicating that the object or value is to be imported. By default, UIL objects and values are private, and are not accessible by other UIL modules. Section 3.2 explains the scope of UIL objects and values in more detail.

2.4.1 String Literals

A string literal is a sequence of zero or more 8-bit or 16-bit characters or a combination of both delimited by apostrophes (') or quotation marks ("). String literals can be no more than approximately 2000 characters long.

A single-quoted string literal can span multiple source lines. To continue a single-quoted string literal, terminate the continued line with a backslash (\ ). The literal continues with the first character on the next line.

Double-quoted string literals cannot span multiple source lines. (Because double-quoted strings can contain escape sequences and other special characters, you cannot use the backslash character to designate continuation of the string.) To build a string value that must span multiple source lines, use the concatenation operator. See Section 2.4.1.3 for a description of how to concatenate strings.

The syntax of a string literal is one of the following:

```
'[ char... ]'

[ #char-set ][ char... ]""
```

Both string forms associate a character set with a string value. The UIL compiler uses the following rules to determine the character set and storage format for string literals:

- A string declared as *'string'* is equivalent to *#ISO_LATIN1"string"*.
- A string declared as *"string"* is equivalent to *#char-set"string"* if you specified *char-set* as the default character set for the module. Otherwise, *"string"* is equivalent to *#ISO_LATIN1"string"*.
- A string of the form *"string"* or *#char-set"string"* is stored as a null-terminated string unless the string consists of mixed 8-bit and 16-bit DEC_HANZI or DEC_KANJI characters (in which case the string is stored as a compound string).
- If a string of the form *#DEC_HANZI"string"* or *#DEC_KANJI"string"* consists of all 8-bit characters, the string is stored as if it were an ISO_LATIN1 string.

- If a string of the form *#DEC_HANZI"string"* or *#DEC_KANJI"string"* consists of all 16-bit characters, the string is stored as a null-terminated string, but retains its character set information in the UID file.

Table 2–5 gives examples of valid and invalid string literal syntax. Note that the COMPOUND_STRING function (described in Section 2.5.6) forces the UIL compiler to generate a compound string.

Table 2–5 Examples of String Literal Syntax

Form	Storage Format	Character Set
'string'	Null-terminated string	ISO_LATIN1
#char-set'string'	Invalid syntax; will not compile	Not applicable
COMPOUND_STRING('string')	Compound string	ISO_LATIN1
"string"	Null-terminated string[1]	Default character set specified for the module; otherwise, ISO_LATIN1
#ISO_GREEK"string"	Null-terminated string[1]	ISO_GREEK
COMPOUND_STRING("string")	Compound string	Default character set specified for the module; otherwise, ISO_LATIN1
COMPOUND_STRING(#ISO_ARABIC"string")	Compound string	ISO_ARABIC
'string'&"string"	If the character sets and writing directions of the operands match, the resulting string is null-terminated; otherwise, the result is a multiple-segment compound string[1]	Character set or sets specified for the individual segments
"string"&#ISO_HEBREW"string"	If the implicit character set and writing direction for the left operand matches the explicit character set (ISO_HEBREW) and writing direction (right to left) for the right operand, the resulting string is a null-terminated string; otherwise, the result is a multiple-segment compound string[1]	Character set or sets specified for the individual segments
'string'&#DEC_KANJI"8-bit string"	Null-terminated string[1]	ISO_LATIN1
'string'&#DEC_KANJI"16-bit string"	Compound string	ISO_LATIN1 for the first segment and DEC_KANJI for the second segment

[1]For the DEC_HANZI or DEC_KANJI character sets, a string consisting of mixed 8-bit and 16-bit characters results in a compound string.

Note that string literals can contain characters with the 8-bit (high-order) set. You cannot type control characters (00 . . . 1F, 7F, and 80 . . . 9F) directly in a single-quoted string literal. However, you can represent these characters with escape sequences. The characters listed in Table 2–6 cannot be directly entered in a UIL module. You must use the indicated escape sequence to enter these characters in a string literal.

Table 2–6 UIL Escape Sequences for String Values

Escape Sequence	Meaning
\b	Backspace
\f	Form feed
\n	New line[1]
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Apostrophe
\"	Quotation mark
\\	Backslash
integer\	Character whose internal representation is given by *integer*, in the range 0 to 255 decimal

[1]For VMS Version 5.3 and 5.4, the UIL compiler does not process newline characters that are embedded in compound strings. The effect of a newline character that is embedded in compound strings is now solely dependent on the character set specified, and the result may not always be the creation of multiline compound strings.

Most XUI Toolkit routines use the null character to determine the end of a string. Therefore, you cannot pass strings with embedded nulls to the Toolkit.

2.4.1.1 Compound String Literals

In addition to its character value, a compound string consist of two properties: an 8-bit or a 16-bit character set and a writing direction. The UIL data type for a compound string is **compound_string**.

These properties are needed to fully understand the value of the string. For example, if the character set of a string is DEC_KANJI (Japanese), the value of the string may contain a mixture of 8-bit and 16-bit codes. Character sets are also important for choosing the font in which to display a character string. If a character string has the ISO_HEBREW character set, a font that contains Hebrew characters should be used to display the string's value.

To create multiline compound strings, you must now create multisegment compound strings with separators. This can be accomplished in UIL with the SEPARATE = *(boolean_expression)* clause of the COMPOUND_STRING function (see Section 2.5.6) and the string concatenation character (&). For example, in VMS Version 5.1, the UIL compiler recognized the following.

```
value
    sample_string : compound_string ("Hello\nWorld!");
```

However, for VMS Version 5.3 and Version 5.4, you must now enter the following:

```
value
    sample_line1 : compound_string ("Hello", separate = true);
    sample_line2 : compound_string ("World!");
    sample_string : sample_line1 & sample_line2;
```

You can still attain VMS Version 5.1 behavior of compound strings by using the /VERSION qualifier when you compile your UIL specification file. The default for the /VERSION qualifier is /VERSION=V2. See Section 4.1.4 for more information on the /VERSION qualifier.

The writing direction of a compound string is implied by the character set specified for the string. You can explicitly set the writing direction for a compound string by using the COMPOUND_STRING function (described in Section 2.5.6). Section 2.4.1.2 describes the character sets supported in UIL for compound strings.

A compound string can consist of a sequence of concatenated strings, each of which can have a different character set property and writing direction. (You can concatenate null-terminated strings with compound strings; the result is a compound string.) Use the concatenation operator (described in Section 2.4.1.3) to create a sequence of compound strings. The following is an example of concatenated compound strings:

```
#ISO_HEBREW"txet werbeh"&#ISO_LATIN8"latin text"
```

Each string in the sequence is stored along with information about the character set and writing direction. You can manipulate a compound string with the XUI Toolkit routines for compound strings.

Generally, a string literal is stored in the UID file as a compound string when the literal consists of concatenated strings having different character sets or writing directions, or when you use the string to specify a value for an argument that requires a compound string value (see Section 2.4.1.4). If you want to guarantee that a string literal is stored as a compound string, you must use the COMPOUND_STRING function described in Section 2.5.6.

2.4.1.2 Character Sets for String Literals

The character sets supported by the UIL compiler are listed in Table 2–7.

The first column shows the UIL name for the character set. The second column indicates whether characters in this set are represented in 8 or 16 bits. The third column gives a brief description of the character set. Note that several UIL names map to the same character set. In some cases, the UIL name influences how string literals are read. For example, strings identified by a UIL character set ending in _LR are read left to right. Names that end in a different number reflect different fonts (for example, ISO_LATIN1 or ISO_LATIN6).

Table 2–7 UIL-Supported Character Sets

UIL Name	Size	Description
ISO_LATIN1	8-bit	GL: ASCII, GR: Latin-1 Supplement
ISO_LATIN2	8-bit	GL: ASCII, GR: Latin-2 Supplement
ISO_ARABIC	8-bit	GL: ASCII, GR: Latin-Arabic Supplement
ISO_LATIN6	8-bit	GL: ASCII, GR: Latin-Arabic Supplement
ISO_GREEK	8-bit	GL: ASCII, GR: Latin-Greek Supplement
ISO_LATIN7	8-bit	GL: ASCII, GR: Latin-Greek Supplement
ISO_HEBREW	8-bit	GL: ASCII, GR: Latin-Hebrew Supplement
ISO_LATIN8	8-bit	GL: ASCII, GR: Latin-Hebrew Supplement
ISO_HEBREW_LR	8-bit	GL: ASCII, GR: Latin-Hebrew Supplement
ISO_LATIN8_LR	8-bit	GL: ASCII, GR: Latin-Hebrew Supplement
JIS_KATAKANA	8-bit	GL: JIS Roman, GR: JIS Katakana
DEC_TECH	8-bit	GL: DEC Spec Graphics, GR: DEC Technical
DEC_KANJI	16-bit	DEC Kanji Character Set (Japanese)
DEC_HANZI	16-bit	DEC Hanzi Character Set (People's Republic of China)

In the following example, the characters in the string value are presented right to left:

```
#ISO_HEBREW"tfel ot thgir morf og sretcarahc"
```

Because the character set for the literal is ISO_HEBREW, the characters in quotation marks can be any legal character as defined by the ISO_HEBREW character set. Table 2–8 lists the parsing rules for each of the character sets.

Table 2–8 Parsing Rules for Character Sets

Character Set	Parsing Rules
All character sets	Character codes in the range 00..1f, 7f, and 80..9f are control characters, including both bytes of 16-bit characters. The compiler flags these as illegal characters.
ISO_LATIN1 ISO_LATIN2 ISO_ARABIC ISO_LATIN3 ISO_GREEK ISO_LATIN4	These sets are parsed from left to right. The escape sequences given for null-terminated strings[1] are also supported by these character sets.

[1]See Table 2–6 for a description of escape sequences for null-terminated strings.

(continued on next page)

Table 2–8 (Cont.) Parsing Rules for Character Sets

Character Set	Parsing Rules
ISO_HEBREW ISO_LATIN8	These sets are parsed from right to left; for example, the string #ISO_HEBREW"012345" generates the string "543210" with character set ISO_HEBREW. The writing direction for this segment is marked as being right_to_left. The escape sequences given for null-terminated strings[1] are also supported by these character sets. The characters that comprise the escape sequence are in left-to-right order. Therefore, you type \n, **not** n\.
ISO_LATIN8_LR	These sets are parsed from left to right, so the string #ISO_LATIN8_LR"012345" generates the string "012345" with character set ISO_LATIN8. The writing direction for this segment is marked as being right_to_left. The escape sequences given for null-terminated strings[1] are also supported by these character sets.
JIS_KATAKANA	This set is parsed from left to right. The escape sequences given for null-terminated strings[1] are also supported by this character set. Note that the backslash (\) can be displayed as a yen symbol.
DEC_KANJI DEC_HANZI	These sets are parsed from left to right. The string can contain a mixture of 8-bit and 16-bit codes; 8-bit codes are in the range 00–7F. A character in the range A0–FF is assumed to be the first byte of a 2-byte character. Among the 8-bit characters, the escape sequences given for null-terminated strings[1] are also supported by these character sets. If a DEC_KANJI or DEC_HANZI literal contains both 8-bit and 16-bit characters, it is a compound string with each sequence of 8-bit characters marked as ISO_LATIN1, and each 16-bit character sequence marked as DEC_KANJI or DEC_HANZI. If a DEC_HANZI or DEC_KANJI literal contains only 8-bit characters, the string is stored as a null-terminated string.

[1]See Table 2–6 for a description of escape sequences for null-terminated strings.

In addition to designating parsing rules for strings, character set information remains an attribute of a compound string. If the character set is included in a compound string consisting of several concatenated segments, the character set information for that string segment is retained. This gives the XUI Toolkit the information it needs to decipher the compound string and to choose a font to display the string.

For an application interface displayed only in English, UIL lets you ignore the distinctions between the two uses of strings. The compiler recognizes by context when a string must be passed as a null-terminated string or as a compound string.

For an application interface displayed in a variety of languages, UIL provides a mechanism for distinguishing the two uses of text and tries to make the UIL specification moderately easy to create with a multilingual editor.

The UIL compiler recognizes enough about the various character sets to correctly parse string literals. The compiler also issues errors if you use a compound string in a context that supports only null-terminated strings.

This means that a mixture of 8-bit and 16-bit characters for DEC_KANJI and DEC_HANZI literals and the right-to-left insertion for ISO_HEBREW literals is permitted. Furthermore, the UIL compiler recognizes the requirements of object arguments, and issues errors if you use a compound string in a context that supports only null-terminated strings.

The character set specification is optional in a string value. If omitted from a compound string, the character set is assumed to be the default for the module as specified in the module declaration (see Section 3.1.3). The default character set for null-terminated strings is ISO_LATIN1 (which is vendor-independent and matches the default font for VMS DECwindows).

Since the character set names are keywords, you must put them in lowercase if case-sensitive names are in force. If names are case insensitive, character set names can be either uppercase, lowercase, or mixed case. (See Section 3.1.2 for more information on case sensitivity for UIL names.)

2.4.1.3 Concatenated String Literals

The concatenation operator (&) takes two strings as operands and creates a new string made up of the left operand followed immediately by the right operand.

For example, **'abcd'** & **'xyz'** becomes **'abcdxyz'**.

The right and left operands of the concatenation operator can be null-terminated strings, compound strings, or a combination. The operands can hold string values of the same or different character sets.

The string resulting from the concatenation is a null-terminated string unless one or more of the following conditions exists:

- One of the operands is a compound string.
- The operands have different character set properties.
- The operands have different writing directions.

If one or more of these conditions exists, the resulting string is a compound string. You cannot use imported or exported values as operands of the concatenation operator. (See Section 3.2 for information on declaring values as private, exported, or imported.)

2.4.1.4 Data Storage Consumption for String Literals

The way a string literal is stored in the UID file depends on how you declare and use the string. The UIL compiler automatically converts a null-terminated string to a compound string if you use the string to specify the value of an argument that requires a compound string. However, this conversion is costly in terms of storage consumption.

Private, exported, and imported string literals require storage for a single allocation when the literal is declared; thereafter, storage is required for each reference to the literal. Literals declared in line require storage for both an allocation and a reference.

Table 2–9 summarizes data storage consumption for string literals. The storage requirement for an allocation consists of a fixed portion and a variable portion. The fixed portion of an allocation is roughly the same as the storage requirement for a reference (a few bytes). The storage consumed by the variable portion depends on the size of the literal value (that is, the length of the string). To conserve storage space, avoid making string literal declarations that result in an allocation per use.

Table 2–9 Data Storage Consumption for String Literals

Declaration	Data Type	Used As	Storage Requirements per Use
In line	Null-terminated	Null-terminated	An allocation and a reference (within the module)
Private	Null-terminated	Null-terminated	A reference (within the module)
Exported	Null-terminated	Null-terminated	A reference (within the UID hierarchy)
Imported	Null-terminated	Null-terminated	A reference (within the UID hierarchy)
In line	Null-terminated	Compound	An allocation and a reference (within the module)
Private	Null-terminated	Compound	An allocation and a reference (within the module)
Exported	Null-terminated	Compound	A reference (within the UID hierarchy)
Imported	Null-terminated	Compound	A reference (within the UID hierarchy)
In line	Compound	Compound	An allocation and a reference (within the module)
Private	Compound	Compound	A reference (within the module)
Exported	Compound	Compound	A reference (within the UID hierarchy)
Imported	Compound	Compound	A reference (within the UID hierarchy)

2.4.2 Integer Literals

An integer literal represents the value of a whole number. Integer literals have the form of an optional sign followed by one or more decimal digits. An integer literal must be in the range –2,147,483,647 to 2,147,483,647. An integer literal must not contain embedded spaces or commas.

Integer literals are stored in the UID file as longwords. Exported and imported integer literals require a single allocation when the literal is declared; thereafter, a few bytes of storage are required for each reference to the literal. Private integer literals and those declared in line require allocation and reference storage per use. To conserve storage space, avoid making integer literal declarations that result in an allocation per use.

Table 2–10 shows data storage consumption for integer literals.

Table 2–10 Data Storage Consumption for Integer Literals

Declaration	Storage Requirements per Use
In line	An allocation and a reference (within the module)
Private	An allocation and a reference (within the module)
Exported	A reference (within the UID hierarchy)
Imported	A reference (within the UID hierarchy)

2.4.3 Boolean Literals

A Boolean literal represents the value true (reserved keyword TRUE or ON) or false (reserved keyword FALSE or OFF). The reserved keywords TRUE and FALSE are subject to case-sensitivity rules.

In a UID file, TRUE is represented by the integer value 1, and FALSE is represented by the integer value 0.

Data storage consumption for Boolean literals is the same as that for integer literals.

2.4.4 Floating-Point Literals

A floating-point literal represents the value of a real (or floating-point) number. Floating-point literals have the form:

[+ | -] digit... . [digit...] [{ E | e } [+ | -] digit...]

or

[+ | -] . digit... [{ E | e } [+ | -] digit...]

A floating-point literal can represent values in the range .29E–38 to 1.7E+38 with up to 16 significant digits. A floating-point literal must not contain embedded spaces or commas.

Floating-point literals are stored in the UID file in D-floating-point format. Table 2–11 gives examples of valid and invalid floating-point notation for the UIL compiler.

Table 2–11 Valid and Invalid Floating-Point Notation

Valid Floating-Point Literals	Invalid Floating-Point Literals
1.0	1e1 (no decimal point)
.1	E–1 (no decimal point or digits)
3.1415E–2 (which equals .031415)	2.87 e6 (embedded blanks)
–6.29e7 (which equals –62900000)	2.0e100 (out of range)

Data storage consumption for floating-point literals is the same as that for integer literals.

2.5 Value-Generating Functions

UIL provides functions to generate the following types of values:

- Colors
- Pixmaps
- Fonts
- Font tables
- Class record names
- Compound strings
- Compound string tables
- ASCIZ string tables
- Integer tables
- Arguments
- Reasons
- Translation tables

Note: In this chapter, all examples assume case-insensitive mode. Keywords are shown in uppercase letters to distinguish them from user-specified names, which are shown in lowercase letters. This use of uppercase letters is not required in case-insensitive mode. In case-sensitive mode, keywords *must* be in *lowercase* letters.

2.5.1 COLOR Function

The COLOR function supports the definition of colors. Using the COLOR function, you can designate a value to specify a color and then use that value for arguments requiring a color value. This function returns a value of type **color**.

The COLOR function has the following syntax:

```
COLOR (string_expression [,FOREGROUND | ,BACKGROUND])
```

The string expression names the color you want to define; the optional keywords FOREGROUND and BACKGROUND identify how the color is to be displayed on a monochrome device when the color is used in the definition of a color table (see Section 2.5.2).

Example 2–1 shows how to use the COLOR function.

In this example, the COLOR function is used with the VALUE declaration (described in Chapter 3) to define three colors and give them each a name. One of these colors, *green*, is then used to specify the foreground color of the main window.

Example 2–1 Using the COLOR Function

```
VALUE red: COLOR( 'Red' );
VALUE green: COLOR( 'Green' );
VALUE blue: COLOR( 'Blue' );

OBJECT primary_window: MAIN_WINDOW
       { ARGUMENTS
           { FOREGROUND_COLOR = green;
             BACKGROUND_COLOR = COLOR( 'Black' );
           };
       };
```

A second use of the COLOR function defines the background color for the main window as the color associated with the string 'Black'.

The UIL compiler does not have built-in color names. Colors are a server-dependent attribute of an object. Colors are defined on each server and may have different RGB values on each server. The string you specify must be recognized by the server on which your application runs.

In a UID file, UIL represents a color as a character string. The XUI Resource Manager (DRM) calls X translation routines that convert a color string to the device-specific pixel value. If you are running on a monochrome server, all colors automatically translate to black or white. If you are on a color server, the color names translate to their proper colors if the following conditions are met:

- The color is defined.
- The color map is not yet full.

If the color map is full, even valid colors translate to black or white (foreground or background).

If you have the VMS DECwindows software installed on your system, you can see a list of the color name strings understood by the VMS DECwindows servers by entering the following command:

```
$ TYPE SYS$MANAGER:DECW$RGB.COM
```

The command procedure DECW$RGB.COM is executed during VMS DECwindows startup to set up the mapping of color names to RGB color indexes. These names are defined so that you can use reasonable names, rather than specifying numeric color levels, to pick colors. (The server stores the equivalent numeric color levels of color names in the file XDEFAULTS.DAT file.)

To write an application that runs on both monochrome and color devices, you need to specify which colors in a color table (defined with the COLOR_TABLE function; see Section 2.5.2) map to the background and which colors map to the foreground. UIL lets you use the COLOR function to designate this mapping in the definition of the color. Example 2–2 shows how to use the COLOR function to map the color red to the background color on a monochrome device.

Example 2–2 Using the COLOR Function to Define Monochrome Mapping for a Color

```
VALUE c: COLOR ( 'red',BACKGROUND );
```

Mapping comes into play only when DRM is given a color and the application is to be displayed on a monochrome device. In this case, each color is considered to be in one of the following three categories:

- The color is mapped to the background color on the monochrome device.
- The color is mapped to the foreground color on the monochrome device.
- Monochrome mapping is undefined for this color.

If the color is mapped to the monochrome foreground or background color, DRM substitutes the foreground or background color, respectively. If you do not specify the monochrome mapping for a color, DRM passes the color string to the XUI Toolkit for mapping to the foreground or background color.

2.5.2 Functions for Specifying Pixmaps

Pixmap values are designed to let you specify labels that are graphic images rather than text. Pixmap values are not directly supported by UIL. Instead, UIL supports icons, which are a simplified form of pixmap. You use a character to describe each pixel in the icon.

You can generate pixmaps in UIL in two ways:

- Define an icon inline using the ICON function (and optionally use the COLOR_TABLE function to specify colors for the icon).
- Use the XBITMAPFILE function, specifying the name of an X bitmap file that you created outside UIL to be used as the pixmap value.

In a UIL module, any argument of type **pixmap** should have an icon or an X bitmap file specified as its value.

2.5.2.1 COLOR_TABLE Function

The COLOR_TABLE function has the following syntax:

```
COLOR_TABLE ( { color_expression = character },... )
```

The color expression is a previously defined color, a color defined inline with the COLOR function, or the phrase BACKGROUND COLOR or FOREGROUND COLOR. The character can be any valid UIL character (see Table 2–1). This function returns a value of type **color_table**.

Example 2–3 shows how to specify a color table.

Example 2–3 Using the COLOR_TABLE Function

```
VALUE
  rgb : COLOR_TABLE ( red = 'r', green = 'g', blue = 'b' );
  bitmap_colors : COLOR_TABLE ( BACKGROUND COLOR = '0', FOREGROUND COLOR = '1' );
```

The COLOR_TABLE function provides a device-independent way to specify a set of colors. The COLOR_TABLE function accepts either previously defined UIL color names or inline color definitions (using the COLOR function). A color table must be private since its contents must be known by the UIL compiler in order to construct an icon. The colors within a color table, however, can be imported, exported, or private (see Section 3.2).

The single letter associated with each color is the character you use to represent that color when creating an icon. See Section 2.5.2.2 for an example. Each letter used to represent a color must be unique within the color table.

2.5.2.2 ICON Function

The ICON function has the following syntax:

ICON ([COLOR_TABLE = color_table_expression ,] row,...)

The color table expression is a previously defined color table and the row is a character expression giving one row of the icon. This function returns a value of type **pixmap**.

Example 2–4 shows how to define a pixmap using the COLOR_TABLE and ICON functions.

Example 2–4 Using the ICON Function

```
VALUE
   rgb     : COLOR_TABLE ( red = '=', green = 'o', blue = ' ' );
   x       : ICON( COLOR_TABLE=rgb, '=========',
                                    '==o   o==',
                                    '== o o ==',
                                    '==  o  ==',
                                    '== o o ==',
                                    '==o   o==',
                                    '=========' );
```

The ICON function describes a rectangular icon that is x pixels wide and y pixels high. The strings surrounded by single quotation marks describe the icon. Each string represents a row in the icon; each character in the string represents a pixel.

The first row in an icon definition determines the width of the icon. All rows must have the same number of characters as the first row. The height of the icon is dictated by the number of rows. For example, the X icon defined in Example 2–4 is 9 pixels wide and 7 pixels high.

The first argument of the ICON function (the color table specification) is optional and identifies the colors that are available in this icon. By using the single letter associated with each color, you can specify the color of each pixel in the icon.

In Example 2–4, an equal sign (=) represents the color red, a lowercase o is green, and a space ( ) is blue. The icon must be constructed of characters defined in the specified color table. In Example 2–4, the color table named *rgb* specifies colors for the equal sign (=), lowercase o, and space. The X icon is constructed with these three characters.

A default color table is used if you omit the argument specifying the color table. To make use of the default color table, the rows of your icon must contain only spaces and asterisks. The default color table is defined as follows:

```
COLOR_TABLE( BACKGROUND COLOR = ' ', FOREGROUND COLOR = '*' )
```

You can specify icons as either private, imported, or exported.

2.5.2.3 XBITMAPFILE Function

The XBITMAPFILE function is similar to the ICON function in that both describe a rectangular icon that is x pixels wide and y pixels high. However, XBITMAPFILE allows you to specify an external file containing the definition of an X bitmap, whereas all ICON function definitions are coded directly within the UIL module. The X bitmap file specified as the argument to the XBITMAPFILE function is read at application run time by DRM.

This function returns a value of type **pixmap** and can be used anywhere a pixmap data type is expected. The XBITMAPFILE function has the following syntax:

```
XBITMAPFILE( string_expression);
```

Example 2–5 shows how to use the XBITMAPFILE function to specify a background pixmap.

Example 2–5 Using the XBITMAPFILE function

```
value
    background_pixmap : xbitmapfile('myfile_button.xbm');
```

In this example, the X bitmap specified in *myfile_button.xbm* is used to create a pixmap, which can be referenced by the value *background_pixmap*.

2.5.3 FONT Function

You define fonts with the FONT function. Using the FONT function, you designate a value to specify a font and then use that value for arguments that require a font value. The UIL compiler has no built-in fonts. Therefore, if you decide to specify fonts in your application interface, you must define all such fonts using the FONT function. This function returns a value of type **font**.

Each font makes sense only in the context of a character set. The FONT function has an additional parameter to let you specify the character set for the font. This parameter is optional; if you omit it, the default character set is ISO_LATIN1. (See Section 2.4.1 for more information on character sets supported by UIL.)

The FONT function has the following syntax:

FONT(string-expression [, CHARACTER_SET = char-set])

The string expression specifies the name of the font (according to the DECwindows font naming convention) and the clause CHARACTER_SET = char_set specifies the character set for the font.

Note: The string expression you use in the FONT function cannot be a compound string.

Example 2–6 shows an example of the FONT function.

Example 2–6 Using the FONT Function

```
VALUE big:  FONT('-ADOBE-Times-Medium-R-Normal--*-140-*-*-P-*-ISO8859-1');
VALUE bold: FONT('-ADOBE-Helvetica-Bold-R-Normal--*-100-*-*-P-*-ISO8859-1');

OBJECT danger_window: CAUTION_BOX
    { ARGUMENTS
        { TITLE = 'You are about to lose all changes';
          FONT_ARGUMENT = FONT_TABLE(bold);
        };
    };
```

In this example, the FONT function is used with the VALUE declaration (described in Chapter 3) to define two fonts and give them each a name. One of these fonts, *bold*, is then converted to a font table using the FONT_TABLE function (since the argument **font_argument** requires a font table) and is used to specify the text font of the caution box. (See Section 2.5.4 for more information on the FONT_TABLE function.)

Use the wildcard character (*) as shown in Example 2–6 to specify fonts in a device-independent manner.

The *VMS DECwindows Xlib Routines Reference Manual* lists the names of fonts supported by the XUI Toolkit, and explains how font names are configured.

Note: In general, you should not specify fonts for objects in your application interface. This allows end users to control font selection through the XDEFAULTS.DAT file.

2.5.4 FONT_TABLE Function

A font table is a sequence of pairs of fonts and character sets. When an object needs to display a string at run time, the object scans the font table for the character set that matches the character set of the string to be displayed. To let you supply such an argument, UIL provides the FONT_TABLE function. This function returns a value of type **font_table**.

The syntax of the FONT_TABLE function is as follows:

FONT_TABLE(font-item,...)

font-item ::= [char-set =] font-expression

The font expression is created with the FONT function.

The font expression specifies the name of the font (according to the DECwindows font-naming convention) and cannot be imported or exported. The optional character set specifies the character set to which this font applies. If you do not specify the character set, the character set specified in the font expression declaration is assumed.

Note: If you specify a font value as an argument that requires a font table value, the UIL compiler automatically converts a font value to a font table.

2.5.5 CLASS_REC_NAME Function

CLASS_REC_NAME is a function that specifies the class name of a widget. It allows you to pass a widget class name as part of an argument. The function returns a value type of **class_rec_name**.

The CLASS_REC_NAME function has the following syntax:

CLASS_REC_NAME(string-expression)

Example 2–7 shows how to use the CLASS_REC_NAME function to specify the class of child widgets that can be added to the menu.

Example 2–7 Using the CLASS_REC_NAME Function

```
work_area_menu
    {
    arguments
        {
        menu_entry_class = CLASS_REC_NAME("togglebuttongadgetclass");
        }
    };
```

2.5.6 COMPOUND_STRING Function

Use the COMPOUND_STRING function to set properties of a null-terminated string and to convert it into a compound string. The properties you can set are the character set and the writing direction. This function returns a value of type **compound_string**.

The COMPOUND_STRING function has the following syntax:

COMPOUND_STRING(string_expression [, property]...);

The result of the COMPOUND_STRING function is a compound string with the string expression as its value. The string expression must be

a null-terminated string. You can optionally include one or more of the following clauses to specify properties for the resulting compound string:

CHARACTER_SET = char-set
RIGHT_TO_LEFT = boolean-expression
SEPARATE = boolean-expression

If you omit the character set clause, the resulting string has the same character set as the string expression. The character sets supported by the UIL are described in Section 2.4.1.2.

The RIGHT_TO_LEFT clause sets the writing direction of the string from right to left if the Boolean expression is true, and left to right otherwise. Specifying this argument does not cause the value of the string expression to change. If you omit the RIGHT_TO_LEFT clause, the resulting string has the same writing direction as the string expression.

The SEPARATE clause appends a separator to the end of the compound string if *boolean-expression* is true. If you omit the SEPARATE clause, the resulting string does not have a separator.

You cannot use imported or exported values as the operands of the COMPOUND_STRING function.

2.5.6.1 Specifying Multiline Compound Strings

In versions of VMS higher than 5.1 (for example, 5.3 and 5.4), the UIL compiler does not consistently process newline characters that are embedded in compound strings. The effect of a newline character embedded in a compound string now depends solely on the character set specified, and the result may not always be the creation of a multiline compound string.

To guarantee the creation of a multiline compound string, you must use the SEPARATE clause in the COMPOUND_STRING function and the concatenation operator (&) to join the segments into a multiline compound string. The SEPARATE clause takes the form SEPARATE = *boolean-expression*, and implements the newline character for VMS Version 5.3 and Version 5.4. For example, in VMS Version 5.1, the UIL compiler would generate a multiline compound string from the following input:

```
value
    sample_string : compound_string ("Hello\nWorld!");
```

To guarantee the same result in VMS Version 5.4, you must type the following:

```
value
    sample_line1 : compound_string ("Hello", separate = true);
    sample_line2 : compound_string ("World!");
    sample_string : sample_line1 & sample_line2;
```

To retain VMS Version 5.1 behavior of the newline character in a compound string, compile your UIL specification file using the /VERSION qualifier as follows:

```
$ UIL/VERSION=V1 MY_FILE.UIL
```

For more information on the /VERSION qualifier, see Section 4.1.4.

2.5.7 COMPOUND_STRING_TABLE Function

A compound string table is an array of compound strings. The names COMPOUND_STRING_TABLE and STRING_TABLE are used as synonyms for this function. Objects requiring a list of string values, such as the **items** and **selected_items** arguments for the list box widget, use string table values. You use the COMPOUND_STRING_TABLE function to build string table values. This function returns a value of type **string_table**.

The COMPOUND_STRING_TABLE function has the following syntax:

COMPOUND_STRING_TABLE(string_expression,...);

Example 2–8 shows how to use the COMPOUND_STRING_TABLE function.

Example 2–8 Using the COMPOUND_STRING_TABLE Function

```
object file_privileges: list_box
    { arguments
        { items = string_table( "owner read",
                                "owner write",
                                "owner delete",
                                "system read",
                                "system write",
                                "system delete",
                                "group read",
                                "group write",
                                "group delete" );
          selected_items = string_table
                               ( "owner read",
                                 "owner write",
                                 "system read",
                                 "system write" );
        };
};
```

Example 2–8 creates a list box with nine menu choices. Four of the choices are initially displayed as having been selected.

The strings inside the string table can be simple strings, which the UIL compiler automatically converts to compound strings if the strings are declared as null-terminated strings..

2.5.8 ASCIZ_STRING_TABLE Function

An ASCIZ string table is an array of ASCIZ (null-terminated) string values separated by commas. The names ASCIZ_TABLE and ASCIZ_STRING_TABLE are used as synonyms for this function. The ASCIZ_STRING_TABLE function allows you to pass more than one ASCIZ string as a callback tag value. This function returns a value of type **asciz_table**.

The ASCIZ_STRING_TABLE function has the following syntax:

ASCIZ_STRING_TABLE(asciz_string_expression,...);

Example 2–9 shows how to use the ASCIZ_STRING_TABLE function to specify a callback tag.

Example 2–9 Using the ASCIZ_STRING_TABLE Function

```
value
    value1 = "my_value_1";
    value2 = "my_value_2";
    value3 = "my_value_3";

object press_my: push_button {
    arguments {
                height = 30;
                width = 10;
        };
    callbacks {
            activate = procedure my_callback(asciz_table(value1, value2, value3));
        };
    };
```

2.5.9 INTEGER_TABLE Function

An integer table is an array of integer values separated by commas. The INTEGER_TABLE function allows you to pass more than one integer tag value per callback reason. This function returns a value of type **integer_table.**

The INTEGER_TABLE function has the following syntax:

INTEGER_TABLE(integer_expression,...);

Example 2–10 shows how to use the INTEGER_TABLE function to define an array of integers to be passed as a callback tag to the procedure *my_callback.*

Example 2–10 Using the INTEGER_TABLE Function

```
value
    value1 = "my_value_1";
    value2 = "my_value_2";
    value3 = "my_value_3";

object press_my: push_button {
    arguments {
                height = 30;
                width = 10;
              };
    callbacks {
        activate = procedure my_callback(integer_table(value1, value2, value3));
     };
};
```

2.5.10 ARGUMENT Function

The ARGUMENT function defines the arguments to a user-defined widget. Each of the objects that can be described by UIL permits a set of arguments, listed in Appendix B. For example, **height** is an argument to most objects and has integer data type. To specify height for a user-defined widget, you can use the built-in argument name **height**, and specify an integer value when you declare the user-defined widget. You do not use the ARGUMENT function to specify arguments that are built into the UIL compiler.

The ARGUMENT function has the following syntax:

```
ARGUMENT( character_expression [, argument_type ] )
```

In this syntax, **character_expression** is the name the UIL compiler uses for the argument in the UID file, while **argument_type** is the type of value that can be associated with the argument. If you omit the second argument, the default type is ANY and no value type checking occurs. Use one of the following keywords to specify the argument type:

- ANY
- ASCIZ_TABLE
- BOOLEAN
- CLASS_REC_NAME
- COLOR
- COLOR_TABLE
- COMPOUND_STRING
- FLOAT
- FONT
- FONT_TABLE
- ICON
- INTEGER
- INTEGER_TABLE
- REASON
- STRING
- STRING_TABLE
- TRANSLATION_TABLE

For example, suppose you built a user-defined widget that draws a circle and takes four arguments: **my_radius**, **my_color**, **x**, and **y**. Example 2–11 shows how to use the ARGUMENT function to define the arguments to this user-defined widget. Note that the ARGUMENT function is not used to specify arguments **x** and **y** because these are built-in argument names. The data type of **x** and **y** is **integer**.

When you declare the *circle* widget, you must use the ARGUMENT function to define the name and data type of the arguments that are not built-ins (*my_radius* and *my_color*). Arguments are specified in an arguments list, identified by the keyword ARGUMENTS in Example 2–11. (See Section 3.5.1 for more information on arguments lists.)

Example 2–11 Using the ARGUMENT Function

```
OBJECT circle:
    USER_DEFINED  PROCEDURE CircleCreate
      { ARGUMENTS
          { ARGUMENT( 'my_radius', INTEGER ) = 1000;
            ARGUMENT( 'my_color', COLOR ) = color_blue;
            x = 1050;
            y = 1050;
          };
      };
```

The following UIL source code is equivalent to that shown in Example 2–11:

```
VALUE circle_radius: ARGUMENT( 'my_radius', INTEGER );
VALUE circle_color: ARGUMENT( 'my_color', COLOR );

OBJECT circle:
    USER_DEFINED  PROCEDURE CircleCreate
      { ARGUMENTS
          { circle_radius = 1000;
            circle_color = color_blue;
            x = 1050;
            y = 1050;
          };
      };
```

In this example, the ARGUMENT function is used in a value declaration (described in Chapter 3) to define the two arguments that are not UIL built-ins: *circle_radius* and *circle_color*. The first of these arguments takes an integer value; the second takes a color value. When referenced in the arguments list of the circle widget, the UIL compiler verifies that the value you specify for each of these arguments is of the type specified in the ARGUMENT function. An example of the arguments list placed in the UID file (and supplied to the creation routine for the circle widget at run time) is given in Table 2–12.

Table 2–12 Values for ARGUMENT Function

Argument	Argument Value
my_radius	1000
my_color	Value associated with the UIL name *color_blue*

You can also use the ARGUMENT function to allow the UIL compiler to recognize extensions to the XUI Toolkit. For example, an existing widget may accept a new argument. Using the ARGUMENT function, you can make this new argument available to the UIL compiler before the updated

version of the compiler is released. Section 3.9 shows an example of this use of the ARGUMENT function.

The *VMS DECwindows Guide to Application Programming* explains how to build and use user-defined widgets in UIL, and shows how DRM constructs the run-time data structures associated with user-defined arguments.

2.5.11 REASON Function

Each of the objects in the XUI Toolkit defines a set of conditions under which it calls a user-defined function. These conditions are known as callback reasons. The user-defined functions are termed callback procedures. In a UIL module, you use a callbacks list to specify which user-defined functions are to be called for which reasons.

The REASON function is useful for defining new reasons for user-defined widgets. When you declare a user-defined widget in UIL, you can define callback reasons for that widget using the REASON function. This function returns a value of type **reason**. The REASON function has the following syntax:

REASON(character_expression)

The character expression specifies the argument name stored in the UID file for the reason. This reason name is supplied to the low-level widget creation routine at run time.

Appendix B lists the callback reasons supported by the XUI Toolkit objects.

Suppose you built a new widget that implements a password system to prevent a set of windows from being displayed unless a user enters the correct password. The widget might define the callbacks shown in Example 2–12.

Example 2–12 Using the REASON Function

```
OBJECT guard_post: USER_DEFINED  PROCEDURE guard_post_create
   { CALLBACKS
        { REASON( 'AccessGrantedCallback' ) = PROCEDURE display_next_level();
          REASON( 'AccessDeniedCallback' ) = PROCEDURE logout();
        };
   };
```

The following UIL source code is equivalent to that shown in Example 2–12:

```
VALUE passed: REASON( 'AccessGrantedCallback' );
VALUE failed: REASON( 'AccessDeniedCallback' );

OBJECT guard_post: USER_DEFINED  PROCEDURE guard_post_create
   { CALLBACKS
        { passed = PROCEDURE display_next_level();
          failed = PROCEDURE logout()};
        };
   };
```

In this example, the REASON function is used in a value declaration to define two new reasons, *passed* and *failed*. The callback list of the widget named *guard_post* specifies the procedures to be called when these reasons occur.

A widget specifies its callbacks by defining a low-level argument for each reason that it supports. The argument to the REASON function gives the name of the low-level argument that supports this reason. Therefore, the low-level argument list placed in the UID file for the widget named *guard_post* includes the arguments arguments listed in Table 2–13:

Table 2–13 Specifying Callbacks Using the REASON Function

Argument Name	Argument Value
AccessGrantedCallback	Callback structure for display_next_level procedure
AccessDeniedCallback	Callback structure for logout procedure

The *VMS DECwindows Guide to Application Programming* shows how DRM constructs the run-time data structures associated with user-defined reasons (which are widget arguments in the XUI Toolkit).

2.5.12 TRANSLATION_TABLE Function

Each of the XUI Toolkit widgets has a translation table that maps events (for example, the left mouse button was pressed) to a sequence of actions. Through widget arguments, such as the **translations** argument, you can specify an alternate set of events or actions for a particular widget.

The TRANSLATION_TABLE function creates a translation table that can be used as the value of an argument that is of the data type **translation_table**. This function returns a value of type **translation_table**. The TRANSLATION_TABLE function has the following syntax:

```
TRANSLATION_TABLE( character_expression,... )
```

Each of the character expressions specifies the run-time binding of an X-level event to a sequence of actions, as shown in Example 2–13. Each of these actions is associated with a procedure that will be executed at run time. Argument lists are discussed in Section 3.5.1.

Example 2–13 defines an argument list called *new_translations*. This arguments list contains the definition of a translation table which is specified as the value for the **translations** argument.

The first line of the definition is a translation table directive that indicates the current translations are to be overridden with those specified in the translation table. In this example, the translations defined by *new_translations* will override the current translations for the *self_destruct* push button.

You can use one of three translation table directives with the TRANSLATION_TABLE function: **#override**, **#augment**, or **#replace** (which is the default). If used, the directive must be the first entry in the translation table, as shown in Example 2–13. Appendix D of the *VMS DECwindows Guide to Application Programming* provides more information on translation tables.

The **#override** directive causes any duplicate translations to be ignored. For example, if <Btn1Down> is already defined in the current translations for a push button, the translation table defined in *new_translations* overrides the current definition. If the **#augment** directive had been specified, the current definition for <Btn1Down> takes precedence over the definition given in *new_translations*. The **#replace** directive replaces all current translations with those specified in the **translations** argument.

Example 2–13 Using the TRANSLATION_TABLE Function

```
LIST new_translations:
     ARGUMENTS
     { TRANSLATIONS =
       TRANSLATION_TABLE
       (
        '#override'
        '<Btn1Down>:          DWTPBFILLHIGHLIGHT() DWTPBARM() DWTPBUNGRAB()',
        '<Btn1Up>:            DWTPBFILLUNHIGHLIGHT() DWTPBACTIVATE() \
                              selection(self_destruct) DWTPBDISARM()',
        '<Btn3Up>:            DWTPBHELP()',
        'Any<LeaveWindow>:    DWTPBFILLUNHIGHLIGHT() DWTPBUNGRAB() DWTPBDISARM()'
       );
     };

OBJECT self_destruct: push_button
     { ARGUMENTS
         { ARGUMENTS new_translations; };
     };
```

2.6 ANY Data Type

The purpose of the ANY data type is to shut off the data type-checking feature of the UIL compiler. You can use the ANY data type for the following:

- Specifying the type of a callback procedure tag
- Specifying the type of a user-defined argument

You can use the ANY data type when you need to use a type not supported by the UIL compiler or when you want the data type restrictions imposed by the compiler to be relaxed. For example, you might want to define a widget having an argument that can accept different types of values, depending on run-time circumstances.

If you specify that an argument takes ANY value, the UIL compiler does not check the type of the value specified for that argument; therefore, you need to take care when specifying a value for an argument of type ANY. You could get unexpected results at run time if you pass a value having a data type that the widget does not support for that argument.

2.7 Include File for Useful Constants

For each of the bindings, there is a file containing useful constants for specifying objects in UIL. Include one of the files in Table 2–14 in your UIL module to have access to the constants.

Table 2–14 UIL Include File Bindings

Binding	File Specification
VAX binding	SYS$LIBRARY:DECW$DWTDEF.UIL
MIT C binding	DECw$Include:DwtAppl.uil

For example, the file DECw$Include:DwtAppl.uil contains the definition of the constant DwtOrientationVertical. You can use this constant for objects that have an **orientation** argument (such as the scroll bar and separator). In the file SYS$LIBRARY:DECW$DWTDEF.UIL, this same constant is named DWT$C_ORIENTATION_VERTICAL.

2.8 Compile-Time Value Expressions

UIL provides literal values for a diverse set of types (integer, string, real, and Boolean) and a set of VMS DECwindows-specific types (for example, colors and fonts). These values are used to provide the value of XUI Toolkit arguments.

UIL allows compile-time value expressions. These expressions can contain references to other UIL values but cannot be forward referenced.

Compile-time value expressions are useful for implementing relative positioning of children without using the attached_dialog_box object type. For example, suppose you want to create a message box inside a dialog box, and you want the message box to be half as wide and half as tall as the dialog box, and centered within it. Using compile-time expressions, you can specify the coordinates of the message box by referring to the values you already defined for the dialog box. If you do not use compile-time value expressions in this case, you would have to compute the x and y location and the height and width of the message box, and recompute these values if the dialog box changes size or location. Furthermore, the computed values would be absolute numbers rather than descriptive expressions like "dialog_box_width / 2".

Note that this example is valid as long as the **units** argument for the dialog box is set to pixels; the width and height of a dialog box are expressed in pixels, but the default unit of measure for positioning within a dialog box is font units. If you do not specify that position argument values (**x** or **y**) are expressed in pixels rather than font units,

expressions using the value of width or height to compute position might give unexpected results.

You can also use compile-time expressions to maintain spacing between objects in a device-independent manner. (The actual measurement of a font unit differs between 75-dpi and 100-dpi monitors.) For example, you can maintain the spacing between a list box and its label by using the following value and object declarations:

```
VALUE  k_delta_y:   11;
       k_y      :   200;
   .
   .
   .
OBJECT listbox_label: LABEL
        { ARGUMENTS { listbox_label_y = k_y; };
        };

OBJECT my_listbox: LIST_BOX
        { ARGUMENTS { listbox_y = k_y + k_delta_y; };
        };
```

In this example, the list box is always 11 font units below its label, regardless of where the label is positioned. Because the height of a label is determined by the height of the font used to display the text of the label (as long as the label object's **conform_to_text** argument is true), you could consider the height of a label to be one (vertical) font unit. Therefore, you obtain roughly the same result regardless of the size of a font unit.

The concatenation of strings is also a form of compile-time expression. Use the concatenation operator to join strings and convert the result to a compound string.

2.8.1 Operators

Table 2–15 lists the set of operators in UIL that allow you to create integer, real, and Boolean values, and to concatenate strings based on other values defined with the UIL module.

Table 2–15 UIL Operators

Operator	Operator Type	Operand Types	Meaning	Precedence
~	Unary	Boolean	NOT	1 (highest)
		integer	1's complement	1
–	Unary	float	negate	1
		integer	negate	1
+	Unary	float	no-op	1
		integer	no-op	1
*	Binary	float, float	multiply	2
		integer, integer	multiply	2

(continued on next page)

Table 2–15 (Cont.) UIL Operators

Operator	Operator Type	Operand Types	Meaning	Precedence
/	Binary	float, float	divide	2
		integer, integer	divide	2
+	Binary	float, float	add	3
		integer, integer	add	3
–	Binary	float, float	subtract	3
		integer, integer	subtract	3
>>	Binary	integer, integer	shift right	4
<<	Binary	integer, integer	shift left	4
&	Binary	Boolean, Boolean	AND	5
		integer, integer	bitwise AND	5
		string[1]	concatenation	5
|	Binary	Boolean, Boolean	OR	6
		integer, integer	bitwise OR	6
^	Binary	Boolean, Boolean	XOR	6
		integer, integer	bitwise XOR	6

[1]String can be either a simple compound string or a sequence of compound strings. If the two concatenated strings have different properties (such as writing direction or character set), the result of the concatenation is a sequence of compound strings.

The result of each operator has the same type as its operands. Note that you cannot mix types in an expression without using conversion routines.

You can use parentheses to override the normal precedence of operators.

In a sequence of unary operators, the operations are performed in right-to-left order. For example, – + –A is equivalent to –(+(–A)).

In a sequence of binary operators of the same precedence, the operations are performed in left-to-right order. For example, A*B/C*D is equivalent to ((A*B)/C)*D.

A value declaration gives a value a name. You cannot redefine the value of that name in a subsequent value declaration.

You can use a value containing operators and functions anywhere you can use a value in a UIL module.

Note: You cannot use exported or imported values as operands in expressions.

2.8.2 Data Type Conversion in Expressions

Several of the binary operators are defined for multiple data types. For example, the operator for multiplication (*) is defined for both floating-point and integer operands. For the UIL compiler to perform these binary operations, both operands must be of the same type. If you supply operands of different data types, the UIL compiler automatically converts

one of the operands to the type of the other according to the conversion rules listed in Table 2–16.

Table 2–16 Automatic Data Type Conversions in UIL

Data Type of Operand 1	Data Type of Operand 2	Conversion Rule
Boolean	Integer	Operand 1 converted to integer
Integer	Boolean	Operand 2 converted to integer
Integer	Floating point	Operand 1 converted to floating point
Floating point	Integer	Operand 2 converted to floating point

You can also explicitly convert the data type of a value by using one of the functions in Table 2–17.

Table 2–17 UIL Data Type Conversion Functions

Function	Result	Comment
INTEGER(boolean)	Integer	TRUE→1, FALSE→0
INTEGER(integer)	Integer	No effect
INTEGER(float)	Integer	Integer part of the floating-point number (truncate toward zero) can result in overflow
FLOAT(boolean)	Floating point	TRUE→1.0, FALSE→0.0
FLOAT(integer)	Floating point	Floating point representation of integer; should not be any loss of precision
FLOAT(float)	Floating point	No effect
COMPOUND_ STRING(string)	Compound string	Converts string to a compound string; no effect if string is already a compound string

Example 2–14 shows a value section from a UIL module containing compile-time expressions and data conversion functions.

Example 2–14 Compile-Time Expressions in a UIL Module

```
VALUE
    outer_box_width:        200;
    outer_box_height:       250;
    box_size_ratio:         0.5;
    inner_box_width:        INTEGER( outer_box_width * box_size_ratio );
    inner_box_height:       INTEGER( outer_box_height * box_size_ratio );
    inner_box_x:            (outer_box_width - inner_box_width) >> 1;
    inner_box_y:            (outer_box_height - inner_box_height) / 2;
    type_field:             0;
    class_field:            16;
    type1:                  1;
    type2:                  2;
    type3:                  3;
    class1:                 1;
    class2:                 2;
    class3:                 3;
    combo1:                 (type1 << type_field) | (class3 << class_field);
    box_name:               'My_Box';    ! Null-terminated string
    box_help_key:           COMPOUND_STRING('Help');  !Compound string
```

3 UIL Module Structure

This chapter describes the structure of a UIL module, explains the scope of references to values and objects defined in UIL, and describes the syntax and use of the following UIL module components:

- Value section
- Procedure section
- List section
- Object section
- Identifier section
- Include directive

In addition, this chapter describes the following features of UIL:

- Definitions for constraint arguments
- Symbolic references to widget identifiers

Note: In this chapter, all examples assume case-insensitive mode. Keywords are shown in uppercase letters to distinguish them from user-specified names, which are shown in lowercase letters. This use of uppercase letters is not required in case-insensitive mode. In case-sensitive mode, keywords *must* be in *lowercase* letters.

3.1 Structure of a UIL Module

A UIL module contains definitions of objects that are to be stored in a **User Interface Definition (UID)** file. A UIL module consists of a module block, which begins with the keyword MODULE and ends with the keywords END MODULE. A module block contains a series of value, procedure, list, identifier, and object sections. There can be any number of these sections in a UIL module. A UIL module can also contain one or more include directives, which can be placed anywhere in the module except within a value, procedure, list, identifier, or object section.

The structure of a UIL module is as follows:

```
uil_module ::=
        MODULE module_name
            [ version_clause ]
            [ case_sensitivity_clause ]
            [ default_character_set_clause ]
            [ default_object_variant_clause ]
```

```
                { value_section
                | procedure_section
                | list_section
                | identifier_section
                | object_section
                | include_directive }...
            END MODULE ";"

version_clause ::=
        VERSION "=" character_expression

case_sensitivity_clause ::=
        NAMES "=" { CASE_SENSITIVE
                    CASE_INSENSITIVE }

default_character_set_clause ::=
        CHARACTER_SET "=" char-set

default_object_variant_clause ::=
        OBJECTS "=" "{" object_type "=" {WIDGET|GADGET}";"..."}"
```

The module name is the name by which this UIL module is known in the UID file. This name is stored in the UID file for later use in the retrieval of resources by the XUI Resource Manager (DRM). This name is always stored in uppercase in the UID file.

Example 3–1 shows the structure of a typical UIL module.

Example 3–1 UIL Module Structure

```
MODULE example                    ! module name (stored as EXAMPLE in UID file)
    VERSION = 'V2.0'              ! version of this module
    NAMES = CASE_INSENSITIVE      ! keywords and names are not case sensitive
    CHARACTER_SET = ISO_LATIN6    ! default character set is ISO_LATIN6
    OBJECTS = { push_button = GADGET; }   ! push buttons are gadgets by default

!+
!       Declare the VALUES, PROCEDURES, LISTS, and OBJECTS
!       here...
!-

END MODULE;
```

The clauses on the module header are described in the following sections.

3.1.1 Version Clause

The version clause specifies the version number of the UIL module. Identify a version clause with the keyword VERSION. The version clause is provided so that an application can guarantee that it is using the correct version of a UIL module. The character expression you use to specify the version can be up to 31 characters in length. In Example 3–1, the version number is V2.0.

3.1.2 Case-Sensitivity Clause

The case-sensitivity clause indicates whether user-specified names in the UIL module are to be treated as case sensitive or case insensitive. For example, if names are case sensitive, then the names "sample" and "Sample" would refer to different components in the UIL module. If names are case insensitive, then "sample" and "Sample" would be treated as the same name and could be used interchangeably to refer to a single component. The default is case insensitive.

Identify a case-sensitivity clause with the keyword NAMES. The case-sensitivity clause must precede any statements that include a user-specified name.

If names are case sensitive in a UIL module, UIL keywords in that module must be in lowercase. Each name is stored in the UID file in the same case as it appears in the UIL module. If names are case insensitive, then keywords can be in uppercase, lowercase, or mixed case, and the uppercase equivalent of each name is stored in the UID file. Table 3–1 summarizes these rules.

Table 3–1 Rules for Case Sensitivity in a UIL Module

Case Sensitivity	Keyword Treatment in UIL Module	Name Treatment in UID File
Case sensitive	Must be entered in lowercase	Names are stored in the same case as they appear in the UIL module
Case insensitive	Can be entered in lowercase, uppercase, or mixed case	Names are stored in uppercase

3.1.3 Default Character Set Clause

The default character set clause specifies the default character set for string literals in the module. Identify the default character set clause with the keyword CHARACTER_SET. The use of the default character set clause is optional.

If specified, the character set clause designates the character set used to interpret an extended string literal if you did not specify a character set for that literal. If you do not include the character set clause in the module header, the default character set for the compilation is ISO_LATIN1. In Example 3–1, the default character set for the module is ISO_LATIN6.

3.1.4 Default Object Variant Clause

The default object variant clause begins with the keyword OBJECTS. Some objects can be either widgets or gadgets. In the object variant clause you list these objects, and specify whether they will be used as a widget or a gadget in this module. A gadget is a simplified version of a widget that consumes fewer resources (and therefore enhances application

performance) but offers limited customization. The following types of user interface objects have both a widget and a gadget variant:

- Label
- Pull-down menu entry
- Push button
- Separator
- Toggle button

For the most part, the widget and gadget variants are interchangeable. To use gadgets in your application, you need to specify to the UIL compiler that you want that particular variant. Otherwise, by default, the UIL compiler assumes you want to use widgets. There are two ways you can specify that you want to use a gadget instead of a widget: You can add a default object variant clause to the module header; or you can add the keyword GADGET to particular object declarations (see Section 3.6).

By using the default object variant clause in the UIL module header, you can declare all labels, pull-down menu entries, push buttons, separators, and toggle buttons, or any combination of these types, to be gadgets. In Example 3–1, only push buttons are declared as gadgets.

To change these objects from one variant to the other, you need only change the default object variant clause. For example, suppose you used the default object variant clause to declare all push buttons as gadgets. To change push button objects from gadgets to widgets, remove the push button type from the clause.

See Section 3.6 for more information on specifying the variant of objects in a UIL module.

3.2 Scope of References to UIL Objects and Values

Resources specified in VALUE or OBJECT sections in a UIL module can have one of the following levels of privacy, or scope:

- Exported—A value or object that you define as exported is stored in the UID file as a named resource, and therefore can be referenced by name in other UID files, or can be fetched from the UID file by the application using DRM literal fetching routines. An exported value or object in UIL is similar to a global variable in a programming language.
- Imported—A value or object that you define as imported is one that is defined as a named resource in a UID file. DRM resolves this value or object declaration with the corresponding exported declaration at application run time. When you define a value or object as imported, DRM looks outside the module in which the imported value or object is declared to get its value at run time. Imported objects do not accept any arguments, controls, or callbacks.

- Private—A private value or object is not imported or exported. A value or object that you define as private is not stored as a distinct resource in the UID file. You can reference a private value or object only in the UIL module containing the declaration. The value or object is directly incorporated into anything in the UIL module that references the declaration. A private value or object in UIL is similar to a local variable in a programming language.

Importing resources is not the same as including resources from another file. The INCLUDE FILE directive (described in Section 3.8) places in line the contents of the specified include file. When you use the INCLUDE FILE directive in a UIL module, the resulting UID file is identical to the one you would obtain if you did not use the INCLUDE FILE directive, but instead directly specified that information in the UIL module. To change an included value, you need to recompile the UIL module.

Importing resources is useful for changing the value of the resource at run time. When you specify a resource as imported, the UID file does not contain the actual value; only the name and data type of the resource are stored. The resource value is not resolved until run time. Since the value, as well as its name and data type, would be stored in the UID file when the UIL module contains only private resources, the UID files would not be identical in this case.

EXPORTED, IMPORTED, and PRIVATE are reserved UIL keywords. By default, values and objects are private.

3.3 Structure of a Value Section

A value section consists of the keyword VALUE followed by a sequence of value declarations, as follows:

```
value_section ::=
        VALUE value_declaration...

value_declaration ::=
        value_name ":"
                { EXPORTED value_expression
                | PRIVATE value_expression
                | value_expression
                | IMPORTED value_type } ";"
```

A value declaration provides a way to name a value expression. The value name can be referred to by declarations which occur later in the UIL module in any context where a value can be used. Values cannot be forward referenced; you must declare a value name before you reference that name.

Value sections can include a clause defining the privacy level for the value. See Section 3.2 for more information about privacy levels in UIL.

Table 3–2 describes the supported value types in UIL.

Table 3–2 UIL Value Types

Value Type	Description
ANY	Prevents the UIL compiler from checking the type of an argument value or callback procedure tag value
ARGUMENT	Defines a value as a user-defined argument
ASCIZ_TABLE	Defines a value as an array of ASCIZ
BOOLEAN	Defines a value as TRUE (ON) or FALSE (OFF)
CLASS_REC_NAME	Defines a value as a class record name
COLOR	Defines a value as a color
COLOR_TABLE	Provides a device-independent way to define a set of colors (usually for a pixmap)
COMPOUND_STRING	Defines a value as a compound string
FLOAT	Defines a value as a floating point literal
FONT	Defines a value as a font
FONT_TABLE	Defines a value as a sequence of font and character set pairs
INTEGER	Defines a value as an integer
INTEGER_TABLE	Defines a value as an array of integers
PIXMAP	Describes a rectangular pixmap using a character to represent each pixel
REASON	Defines a condition under which an object is to call an application function
STRING	Defines a value as a primitive string
STRING_TABLE	Defines a value as an array of compound strings
TRANSLATION_TABLE	Defines an alternative set of events or actions for a widget

Example 3–2 shows how to declare values.

Example 3–2 UIL Value Declaration

```
VALUE
    k_main                  : EXPORTED 1;
    k_main_menu             : EXPORTED 2;
    k_main_command          : EXPORTED 3;

VALUE
    white               : IMPORTED COLOR;
    blue                : IMPORTED COLOR;
    arg_name            : PRIVATE 'new_argument_name';

VALUE
    main_prompt         : 'next command';          ! PRIVATE by default
    main_font           : IMPORTED FONT;
```

Because the values *k_main*, *k_main_menu*, and *k_main_command* are defined as exported and are integers, you can use these values in another UIL module as follows:

```
VALUE
    k_main                  : IMPORTED integer;
```

3.4 Structure of a Procedure Section

A procedure section consists of the keyword PROCEDURE followed by a sequence of procedure declarations, as follows:

```
procedure_section ::=
        PROCEDURE procedure_declaration...

procedure_declaration ::=
        procedure_name
            [ formal_parameter_spec ]";"

formal_parameter_spec ::=
            "(" [ value_type ] ")"
```

Use a procedure declaration to declare a routine that can be used as a callback routine for an object or as the creation routine for a user-defined widget (see Example 2–12). You can reference a procedure name in declarations that occur later in the UIL module in any context where a procedure may be used. Procedures cannot be forward referenced; you must declare a procedure name before you reference it. You cannot use a name you used in another context as a procedure name.

In a procedure declaration, you have the option of specifying that a parameter will be passed to the corresponding callback routine at run time. (This parameter is called the callback tag.) You can specify the data type of the callback tag by putting the data type in parentheses following the procedure name. When you compile the module, the UIL compiler checks that the argument you specify in references to the procedure is of this type. Note that the data type of the callback tag must be one of the valid UIL value types (see Table 3–2).

If you explicitly include a parameter specification in the procedure declaration, the UIL compiler checks the argument type when the procedure is referenced. For example, in the following procedure declaration, the callback procedure named *toggle_proc* will be passed an integer tag at run time. The UIL compiler checks that the parameter specified in any reference to procedure *toggle_proc* is an integer.

```
PROCEDURE
        toggle_proc    (integer);
```

While it is possible to use any UIL data type to specify the type of a tag in a procedure declaration, you must be able to represent that data type in the high-level language you will be using to write your application program. Some data types (such as integer, boolean, and string) are common data types recognized by most programming languages. Other UIL data types (such as string tables) are more complicated and may require you to set up an appropriate corresponding data structure in the application in order to pass a tag of that type to a callback procedure.

Table 3–3 summarizes how the UIL compiler checks argument type and argument count, depending on the procedure declaration.

Table 3–3 UIL Compiler Rules for Checking Argument Type and Count

Declaration	Description of Rule
No parameters	No argument type or argument count checking occurs. You can supply one argument or no arguments in the procedure reference.
()	Checks that the argument count is zero.
(ANY)	Checks that the argument count is one. Does not check the argument type. Use the ANY type to prevent type checking on procedure tags.
(value_type)	Checks for one argument of the specified value type.

You can also use a procedure declaration to specify the creation routine for a user-defined widget. In this case, you specify no formal parameters. The procedure is invoked with the standard four arguments passed to all low-level widget creation routines. (See the *VMS DECwindows Toolkit Routines Reference Manual* for more information about low-level widget creation routines.)

Example 3–3 shows how to declare procedures.

Example 3–3 UIL Procedure Declaration

```
PROCEDURE
    app_help (INTEGER) ;
    app_destroy (INTEGER) ;
```

3.5 Structure of a List Section

A list section consists of the keyword LIST followed by a sequence of list declarations, as follows:

```
list_section ::=
        LIST list_declaration...

list_declaration ::=
        list_name ":" list_definition ";"

list_definition ::=
        list_type list_spec
list_type ::=
        { ARGUMENTS
        | CALLBACKS
        | CONTROLS
        | PROCEDURES }
list_spec ::=
        { list_name
        | "{" list_entry... "}" }

list_entry ::=
        { list_definition
        | argument_list_entry
        | control_list_entry
        | callback_list_entry } ";"
```

You use lists to group together a set of arguments, controls (children), or callbacks for later use in the UIL module. Lists can contain other lists, so that you can set up a hierarchy to clearly show which arguments, controls, and callbacks are common to which objects. You cannot mix the different types of lists; that is, a list of a particular type cannot contain entries of a different list type or reference the name of a different type of list.

A list name is always private to the UIL module in which you declare the list and cannot be stored as a named resource in a UID file.

There are three types of UIL lists, as follows:

- Arguments list, having the list type ARGUMENTS
- Callbacks list, having the list type CALLBACKS
- Controls list, having the list type CONTROLS

These list types are described in the following sections.

3.5.1 Arguments List Structure

An arguments list defines which arguments are to be specified in the argument list parameter when the creation routine for a particular object is called at run time. An arguments list also specifies the values for those arguments. Each arguments list entry has the following syntax:

```
argument_list_entry ::=
        argument_name "=" value_expression
```

The argument name must be either a built-in argument name or a user-defined argument name that is specified by using the ARGUMENT function (see Section 2.5.10). If you use a built-in argument name, the type of the value expression must match the allowable type for this argument.

If you use a built-in argument name as an arguments list entry in an object definition, the UIL compiler checks the argument name to be sure that it is supported by the type of object that you are defining. If the same argument name appears more than once in a given arguments list, the last entry that uses that argument name supersedes all previous entries with that name, and the compiler issues a diagnostic message.

Some arguments are coupled by the UIL compiler. When you specify one of the arguments, the compiler also sets the other. The coupled argument is not available to you. The coupled arguments are listed in Table 3–4.

Table 3–4 UIL Coupled Arguments

Supported Argument	Coupled Argument
items	items_count
selected_items	selected_items_count

See Appendix B for information about supported arguments, their data types, and default values for each widget in the XUI Toolkit.

Example 3–4 shows how to declare an arguments list.

Example 3–4 UIL Arguments List Declaration

```
LIST
    default_size: ARGUMENTS {
        HEIGHT = 500;
        WIDTH = 700;
    };

    default_args: ARGUMENTS {
        ARGUMENTS default_size;
        FOREGROUND_COLOR = white;
        BACKGROUND_COLOR = blue;
    };
```

3.5.2 Callbacks List Structure

Use a callbacks list to define which callback reasons are to be processed by a particular object at run time. Each callbacks list entry has the following syntax:

```
callback_list_entry ::=
        reason_name "=" procedure_reference

procedure_reference ::=
        PROCEDURE procedure_name
            [ "(" [ value_expression ] ")" ]
```

A UIL built-in reason maps to a callback argument in the XUI Toolkit. For XUI Toolkit objects, the reason must be a built-in reason. For a user-defined widget, you can use a reason that you previously declared using the REASON function (see Chapter 2). If you use a built-in reason in an object definition, the UIL compiler ensures that the reason is supported by the type of object you are defining. Appendix B contains information about which callback reasons are supported by which objects.

If the same reason appears more than once in a callbacks list, the last entry that uses that reason supersedes all previous entries using that reason, and the UIL compiler issues a diagnostic message.

You must have previously declared the procedure name you use in a callbacks list. If you specified a value for the callback tag, the tag type must match the value type of the parameter in the corresponding procedure declaration. See Section 3.4 for a detailed explanation of argument type checking for procedures.

Example 3–5 shows how to declare a callbacks list.

Example 3–5 UIL Callbacks List Declaration

```
LIST
    default_callbacks : CALLBACKS {
        destroy = PROCEDURE app_destroy (k_main);
        help    = PROCEDURE app_help (k_main);
    };
```

The following lines of pseudocode show the interface to the callback procedure:

```
PROCEDURE procedure-name ( widget by reference,
                           tag by reference,
                           object-specific structure by reference )

        RETURNS: no_value;
```

Since the UIL compiler produces a UID file rather than an object module (.OBJ), the binding of the UIL name to the address of the entry point to the procedure is not done by the VMS Linker. Instead, the binding is established at run time using the DRM routine REGISTER DRM NAMES. You call this routine prior to fetching any objects, giving it both the UIL name and the procedure address of each callback. The name you register with DRM in the application program must match the name you specified in the UIL module.

Each callback procedure receives three arguments. The first two arguments have the same form for each callback. The form of the third argument varies from object to object.

The first argument is the address of the data structure maintained by the XUI Toolkit for this object instance. This address is called the widget ID for this object.

The second argument is the address of the value you specified in the callbacks list for this procedure. For example, the **destroy** callback in Example 3–5 has *app_destroy* as its callback procedure. The second argument is the address of the integer *k_main*. If you do not specify an argument, the address is null.

Consult the *VMS DECwindows Toolkit Routines Reference Manual* (which describes each object) to find the structure of the third argument. A code representing the reason name you specified in the UIL module is the first field in this structure. The XUI Toolkit encoding of the UIL reasons is contained in the file DECW$DWTDEF.UIL (in the VAX binding) or DwtAppl.uil (in the MIT C binding).

3.5.3 Multiple Procedures per Callback Reason

You can specify multiple procedures for each callback reason in UIL by defining the procedures as a type of list. Just as with other list types, you can define procedures lists either inline, or in a separate list section that you reference by name.

If you define a reason more than once (for example, when the reason is defined both in a referenced procedures list and in the callbacks list for an object), all previous definitions are overridden by the latest definition.

The following is the syntax for specifying multiple procedures per callback reason:

```
PROCEDURES procedure_list_spec

procedure_list_spec ::=
        { procedure_list_name
        | "{" [procedure_list_clause...] "}" }

procedure_list_clause ::=
        { procedure_list_specification | procedure_list_ref }

procedure_list_ref ::]
        procedure_list_name [ "(" [ value_expression ] ")" ]

callback_list_entry ::=
        reason_name "=" procedure_list_spec
```

Example 3–6 shows how to specify multiple procedures per callback reason defined inline in an object declaration for the push_button gadget.

Example 3–6 Specifying Multiple Procedures per Callback Reason

```
object m_quit_button: push_button {
    arguments {
                    .
                    .
                    .
                };
      callbacks {
        activate = procedures
              {
               quit_proc ( 'normal demo exit' ); /* First proc for activate reason */
               shutdown ();                      /* Second proc for activate reason */
              };
      };
};
```

3.5.4 Controls List Structure

A controls list defines which objects are children of, or controlled by, a particular object. Each controls list entry has the following syntax:

```
control_list_entry ::=
        [ MANAGED | UNMANAGED ] object_definition
```

If you specify the keyword MANAGED, at run time the object is created and managed; if you specify UNMANAGED, the object is created only at run time. Objects are managed by default.

Unlike the arguments list and the callbacks list, a controls list entry that is identical to a previous entry does not supersede the previous entry. At run time, each controls list entry causes a child to be created when the parent is created. If you repeat the same object multiple times in a controls list, multiple instances of the child are created at run time.

See Appendix B for a list of the object types a widget can control.

Example 3–7 shows how to declare a controls list.

Example 3–7 UIL Controls List Declaration

```
LIST
    default_main_controls : CONTROLS {
            COMMAND_WINDOW main_command;
            MENU_BAR main_menu;
            UNMANAGED LIST_BOX file_menu;
            UNMANAGED OPTION_MENU edit_menu;
    };
```

3.6 Structure of an Object Section

An object section consists of the keyword OBJECT followed by a sequence of object declarations, as follows:

```
object_section ::=
        OBJECT object_declaration...

object_declaration ::=
        object_name ":"
                { EXPORTED object_definition
                | PRIVATE object_definition
                | object_definition
                | IMPORTED object_type } ";"

object_definition ::=
        object_type [ procedure_reference ] [WIDGET | GADGET] object_spec

procedure_reference ::=
        PROCEDURE procedure_name
            [ "(" [ value_expression ] ")" ]

object_spec ::=
        { object_name
        | "{" list_definition... "}" }
```

Use an object declaration to define the objects that are to be stored in the UID file. You can reference the object name in declarations that occur elsewhere in the UIL module in any context where an object name can be used (for example, in a controls list, as a symbolic reference to a widget ID, or as the **tag_value** argument for a callback procedure). Objects can be forward referenced; that is, you can declare an object name after you reference it. All references to an object name must be consistent with the type of object, as specified in the object declaration. You cannot use a name you used in another context as an object name.

You can specify an object as exported, imported, or private (see Section 3.2).

The object definition contains a sequence of lists that define the arguments, widget hierarchy, and callbacks for the widget. You can specify only one list of each type for an object (see Section 3.5). If you want to specify more than one list of arguments, controls, or callbacks, you can do so within one list, as follows:

```
object some_widget:
    arguments {
        ARGUMENTS arguments_list1;
        ARGUMENTS arguments_list2;
    };
```

In this example, *arguments_list1* and *arguments_list2* are lists of arguments that were previously defined in a LIST section.

You must include a procedure reference to the widget creation routine when declaring a user-defined object. The procedure you specify in the procedure reference is the creation routine for the user-defined object. (The *VMS DECwindows Guide to Application Programming* explains how to build a user-defined object.)

See Appendix B for a list of the supported object types.

Example 3–8 shows how to declare an object.

Example 3–8 UIL Object Declaration

```
OBJECT
    app_main : EXPORTED MAIN_WINDOW {

        ARGUMENTS {
            ARGUMENTS default_args;
            HEIGHT = 1000;
            WIDTH = 800;
        };

        CALLBACKS default_callbacks;

        CONTROLS {
            MENU_BAR main_menu;
            USER_DEFINED my_object;
        };
    };
```

If you want to use gadgets only in certain instances of an object, explicitly specify that particular object as a gadget in its declaration, and omit that object type from the objects clause on the module header (see Section 3.1.4). For example, if you want all push buttons but one to be widgets, omit the push button type from the objects clause and declare the one exception as a gadget in its object section.

Since you do not specify the variant for imported objects, the variant (that is, whether a widget or gadget) of imported objects is unknown until run time.

The following sections detail UIL syntax for specifying object variants, show an example of a UIL module in which gadgets are specified (using both the objects clause and explicit declaration methods), and describe UIL compiler diagnostics related to gadgets.

3.6.1 Specifying Object Variant in the Module Header

You can include a default object variant clause in the module header to specify the default variant of objects defined in the module, on a type-by-type basis.

The object type can be any user interface object type that has a gadget variant (separator, label, push button, or toggle button). If you specify any other object type as a gadget, the UIL compiler issues a diagnostic.

When you include an object type in the objects clause, all objects of that type default to the variant you specified in the objects clause. For example, the following objects clause specifies that all push buttons in the current module are gadgets:

```
OBJECTS = { push_button = GADGET; }
```

The UIL compiler issues an informational diagnostic if you attempt to specify an object type more than once in the objects clause.

You can override the specification you made in an objects clause when you declare a particular object. If you omit the objects clause, or omit an object type from the objects clause, the UIL compiler assumes you want the omitted type to be widget. You can also explicitly override this default in an object declaration. Example 3–9 shows how to use the objects clause and how to override the variant specification in an object declaration.

3.6.2 Specifying Object Variant in the Object Declaration

You can use one of the keywords WIDGET or GADGET as an attribute of an object declaration. You include the keyword between the object type and the left brace of the object specification. Use the GADGET or WIDGET keyword to specify the object type or to override the default variant for this object type. (The UIL compiler assumes all interface objects are widgets by default. You can specify the gadget variant as the default for labels, separators, push buttons, and toggle buttons in an objects clause on the module header, as described in Section 3.6.1.)

The object type can be any user interface object type that has a gadget variant (separator, label, push button, or toggle button). If you specify any other object type as a gadget, the UIL compiler issues a diagnostic.

Example 3–9 shows how to specify gadgets in UIL.

Example 3–9 Specifying User Interface Object Variants

```
MODULE sample
    NAMES = case_insensitive
    OBJECTS =
        { separator = GADGET; push_button = WIDGET; }

    OBJECT
        a_button : push_button GADGET {
            ARGUMENTS { label_label = 'choice a'; };
        };

        a_menu : pulldown_menu {
            ARGUMENTS { border_width = 2; };
            CONTROLS {
                push_button a_button;
                separator GADGET {};
                push_button {
                    ARGUMENTS { label_label = 'choice b'; };
                };
                separator WIDGET {};
                push_button c_button;
                separator {};
            };
        };

        c_button : push_button GADGET {
            ARGUMENTS { label_label = 'choice c'; };
        };

END MODULE;
```

In Example 3–9, the objects clause specifies that all separator objects are gadgets and all push button objects are widgets, unless overridden. Object *a_button* is explicitly specified as a gadget. Object *a_menu* defaults to a widget.

Notice that the reference to *a_button* in the control list of *a_menu* refers to the *a_button* gadget; you need to include the gadget attribute only on the declaration of *a_button*, not on each reference to *a_button*. The same holds true for *c_button*, even though the reference to *c_button* in the control list for *a_menu* is a forward reference.

The unnamed push button definition in the control list for *a_menu* is a widget because of the objects clause; the last separator is a gadget for the same reason.

You need to specify the GADGET or WIDGET keyword only in the declaration of an object, not when you reference the object. You cannot specify the GADGET or WIDGET keyword for a user-defined object; user-defined objects are always widgets.

3.7 Structure of an Identifier Section

Identifiers provide run-time binding of data to names that you specify in a UIL module. Identifiers work like global variables in a programming language.

3.7 Structure of an Identifier Section

List the names of identifiers in an identifier section in a UIL module. An identifier section consists of the reserved keyword IDENTIFIER followed by a list of names, with each name followed by a semicolon. You can use these names later in the UIL module as either the value of an object argument or as the tag value to a callback procedure. At run time, use the DRM routine REGISTER DRM NAMES to bind the identifer name with the data associated with the identifier. (See the *VMS DECwindows Toolkit Routines Reference Manual* for information about the REGISTER DRM NAMES routine.)

Because UIL has a single name space, you cannot use a name you used for a value, object, or procedure as an identifier name.

Your application can successively call the routine REGISTER DRM NAMES with the same identifier names to supersede the value of that name for all subsequent calls to DRM that might use these identifiers. For example, you would use this procedure to change callback tags for objects created from a template definition. (Refer to the *VMS DECwindows Guide to Application Programming* for information on using an object definition as a template.)

Example 3–10 shows an identifier section in a UIL module.

Example 3–10 Using Identifiers in a UIL Module

```
MODULE id_example
    NAMES = CASE_INSENSITIVE

        IDENTIFIER
            my_x_id;
            my_y_id;
            my_focus_id;

        PROCEDURE
            my_focus_callback ( STRING );

        OBJECT my_main : MAIN_WINDOW {
            ARGUMENTS {
                x = my_x_id;
                y = my_y_id;
            };

            CALLBACKS {
                focus = PROCEDURE my_focus_callback ( my_focus_id );
            };
        };

END MODULE;
```

The UIL compiler does not do any type checking on the use of identifiers in a UIL module. Unlike a UIL value, an identifier does not have a UIL type associated with it. You can use an identifier as an object argument or callback procedure tag, regardless of the data type specified in the object or procedure declaration.

To reference these identifier names in a UIL module, use the name of the identifier wherever you want its value to be used. The value is determined at run time. The UIL module in Example 3–10 shows identifiers used as argument values and callback procedure tags. However, you can reference an identifier in any context where you can reference a value.

The identifiers *my_x_id* and *my_y_id* are used as argument values for the main window widget, *my_main*. The position of the main window widget may depend on the screen size of the terminal on which the interface is displayed. Using identifiers, you can provide the values of the **x** and **y** arguments at run time.

The identifier named *my_focus_id* is specified as the tag to the callback procedure *my_focus_callback*. In the application program, you could allocate a data structure and use *my_focus_id* to store the address of that data structure. When the **focus** reason occurs, the data structure is passed as the tag to procedure *my_focus_callback*.

3.8 The UIL Include Directive

The UIL include directive incorporates the contents of a specified file into a UIL module. This mechanism allows several UIL modules to share common definitions. The syntax for the include directive is as follows:

```
include_directive ::=
        INCLUDE FILE character_expression ";"
```

The file specified in the include directive is called an include file. The UIL compiler replaces the include directive with the contents of the include file and processes it as if these contents had appeared in the current UIL source file.

An include file can contain include directives; therefore, include files can be nested. The UIL compiler can process up to 100 references (including the file containing the UIL module), allowing you to include up to 99 files in a single UIL module (including nested files). Each file opening counts as a reference: including the same file twice counts as two references.

The character expression is a file specification that identifies the file to be included. The rules for finding the specified file are similar to the rules for finding header, or .h, files using the include directive, #include, with a quoted string in the VAX C language.

If you do not supply a file extension, .UIL is assumed. If you do not supply a directory, the UIL compiler searches for the include file in the directory of the main source file; if the compiler does not find the include file there, the compiler looks in the directory UIL$INCLUDE (if this logical name has been defined on your system). If you supply a directory, the UIL compiler searches only that directory for the file.

If you use a search list to specify the directory for the main UIL source file, then the compiler uses that search list to locate the included file as well; the UIL compiler searches all directories in the list.

Example 3–11 shows an example of the include directive.

Example 3–11 UIL Include Directive

```
!+
!    The .UIL extension is provided by default.
!-

    INCLUDE FILE 'constants';
```

3.9 Definitions for Constraint Arguments

The XUI Toolkit and the X Toolkit (intrinsics) directly support constraint arguments. A constraint argument is one that is passed directly to children of an object, beyond those arguments that are normally available. For example, the attached dialog box widget is an object that grants a set of constraint arguments to its children. These constraint arguments are used to control the position of the children in the attached dialog box.

Because UIL did not directly support constraint arguments in VMS Version 5.1, you had to define constraint arguments using the ARGUMENT function, as shown in the following example:

```
value
    adb_top_attachment: argument( 'abdTopAttachment', integer);
```

Unlike the arguments that define the attributes of a particular widget, constraint arguments are used exclusively to define the attachments of the children of a particular widget. To supply arguments to a child object, you include the constraint arguments in the arguments list for the child object, as shown in Example 3–12:

Example 3–12 Defining Constraint Arguments

```
object my_dialog_box: dialog_box {
    arguments {
                x = 70;
                y = 20;
                row = 35;
              };
       controls {
          push_button {
               arguments {
                           adb_left_attachment = DwtATTACH_WIDGET;
                           adb_left_offset = 10;
                         };
                  };
         };
};
```

3.10 Symbolic References to Widget Identifiers

The UIL compiler allows you to symbolically refer to a widget identifier by using its name. This mechanism addresses the problem that the UIL compiler views objects by name and the XUI Toolkit views objects by widget identifier. Widget identifiers are defined at run time and are therefore unavailable for use in a UIL module.

When you need to supply an argument that requires a widget identifier, you can use the UIL name of that widget and its object type as the argument. For example, the menu bar widget has an argument **DwtNMenuHelpWidget** that expects the identifier of a widget (a pull-down menu entry, for instance). You can give the name and object type of the pull-down menu entry widget as the value for this argument. Another practical use of a symbolic reference is to specify the default push button widget (in a dialog box widget or radio box widget).

Note: To completely specify a symbolic reference in UIL, you must include the object type with the object name.

Example 3–13 shows the use of a symbolic reference.

Example 3–13 Using Symbolic References in a UIL Module

```
MODULE symbolic_ref_example
    NAMES = CASE_INSENSITIVE

        OBJECT my_dialog_box : DIALOG_BOX {
            ARGUMENTS {
                default_button = PUSH_BUTTON yes_button;
            };

            CONTROLS {
                PUSH_BUTTON yes_button;
                PUSH_BUTTON no_button;
            };
        };

        OBJECT yes_button : PUSH_BUTTON {
            ARGUMENTS {
                label_label = 'yes';
            };
        };

        OBJECT no_button : PUSH_BUTTON {
            ARGUMENTS {
                label_label = 'no';
            };
        };

END MODULE;
```

In Example 3–13, two push button widgets are defined, named *yes_button* and *no_button*. In the definition of the dialog box widget, the name *yes_button* is given as the value for the **default_button** argument. Usually, the default push button argument accepts a widget identifier. When you use a symbolic reference (the object type and name of the *yes_button* widget) as the value for the default push button argument, DRM substitutes the widget identifier of the *yes_button* push button for its name at run time.

3.10 Symbolic References to Widget Identifiers

There is a restriction on the use of symbolic references: the object name you reference must be a descendant of the object being fetched in order for DRM to find the referenced object; you cannot reference an arbitrary object. DRM checks this at run time.

The UIL built-in tables listed in Appendix B indicate where symbolic referencing of widget identifiers is acceptable by showing the term *object reference* as the type of an argument.

4 Using the UIL Compiler

This chapter describes the following:

- Invoking the compiler
- Using include files
- Interpreting diagnostics issued by the compiler
- Interpreting the listing produced by the compiler

4.1 Invoking the UIL Compiler

You invoke the UIL compiler with the DCL command UIL. The UIL command takes a single parameter, which is the name of the file containing the UIL module you want to compile.

The syntax for the UIL command is as follows:

```
UIL [ /qualifier... ] input-file-spec
```

If you do not specify a directory, the input file is assumed to be in the current default directory. If you omit the file type, the type UIL is assumed.

You can use the qualifiers listed in Table 4–1 on the UIL command line.

Table 4–1 Command Line Qualifiers for the UIL Compiler

Command Qualifiers	Default
/[NO]LIST[=file-spec]	/NOLIST for interactive mode /LIST for batch mode
/[NO]MACHINE_CODE	/MACHINE_CODE
/[NO]OUTPUT[=file-spec]	/OUTPUT
/VERSION[=V1 | V2]	/VERSION=V2
/[NO]WARNINGS[=(message_type)]	/WARNINGS

4.1.1 /LIST Qualifier

The /LIST qualifier directs the compiler to produce a listing file. In interactive mode, the default for this qualifier is /NOLIST. In batch mode, the default for this qualifier is /LIST.

When /LIST is in effect, the compiler, by default, creates a listing file in the current default directory with the same name as the input file and the LIS file type. If you include a file specification with the /LIST qualifier, the compiler uses that specification to name the listing file.

4.1.2 /MACHINE_CODE Qualifier

The /MACHINE_CODE qualifier works like the /MACHINE_CODE (or equivalent) qualifier available for most high-level language compilers. If you specify both /LIS and /MACHINE_CODE on the UIL command line, the compiler places in the listing file a description of the records that it wrote to the UID file. This is analogous to a compiler placing the machine code it generates for a high-level language in the listing.

The primary purpose of this listing is to isolate errors in the UIL compiler.

The default is /NOMACHINE_CODE.

4.1.3 /OUTPUT Qualifier

The /OUTPUT qualifier directs the compiler to produce a User Interface Definition (UID) file. By default, /OUTPUT creates a UID file in the current default directory with the same name as the input file and with the UID file type. If you include a file specification with the /OUTPUT qualifier, the compiler uses that specification to name the UID file.

The compiler does not produce a UID file when you use the /NOOUTPUT qualifier or when the compiler issues any diagnostics categorized as error or severe. You can use the /NOOUTPUT qualifier when you want to verify only that your UIL module is syntactically correct.

4.1.4 /VERSION Qualifier

The /VERSION qualifier provides upward compatibility between Version 1.0 and Version 2.0 of the UIL compiler. In particular, the /VERSION qualifier allows you to continue building interfaces that will run under the XUI Toolkit in VMS Version 5.1 (for example, the processing of newline characters that are embedded in compound strings), while still being able to use the new UIL compiler features implemented for VMS Version 5.4.

Allowable values for the /VERSION qualifier are V1 (for VMS Version 5.1) and V2 (for VMS Version 5.3 and 5.4). The default is /VERSION=V2.

4.1.5 /WARNINGS Qualifier

The /WARNINGS qualifier directs the UIL compiler to suppress warning or informational messages or both. Table 4–2 describes the supported values for the /WARNINGS qualifier.

Table 4–2 Values for the /WARNINGS Qualifier

Qualifier	Description
/WARNINGS	Generates all diagnostic messages
/WARNINGS=(NOINFORMATIONALS)	Suppresses informational messages
/WARNINGS=(NOWARNINGS)	Suppresses warning messages
/WARNINGS=(NOWARNINGS,NOINFORMATIONALS)	Suppresses warning and informational messages
/NOWARNINGS	Suppresses warning and informational messages

If the /WARNINGS qualifier is not specified, the UIL compiler generates all diagnostic messages.

4.2 Getting Online Help for the UIL Compiler

You can access online help for the UIL compiler by entering the following command:

```
$ HELP UIL
```

Online help for the UIL compiler includes the following topics:

- Description of the UIL command parameter and qualifiers
- Description of and corrective action for compile-time errors
- How to use the compiler
- Summary of the steps needed to integrate a UIL specification with an application program

4.3 Interpreting Compiler Diagnostics

The UIL compiler issues standard VMS diagnostics to the file SYS$ERROR. The following example shows the form of these messages:

```
        value d: font( 1 );
                       *
%UIL-E-WRONG_TYPE, found integer value when expecting string value
         line: 9  file: DISK$:[PROJECT.UIL.WORK]VALUE_ERROR.UIL;7
```

The first line is the source line that produced the diagnostic. If for some reason the compiler cannot retrieve the source line from the input file (for example, the file is not a disk file), this line is omitted. Any control characters in the text of the line are printed as question marks (?).

The second line of the diagnostic marks the start of the construct that resulted in the diagnostic. In this example, the literal 1 is marked with an asterisk. If the error is not associated with a particular construct, the second line is omitted. If the source line cannot be retrieved, this line is also omitted. The column information is included with the line and file information following the diagnostic message.

The third line is the diagnostic issued by the UIL compiler.

The fourth line specifies the file containing the source line being diagnosed and the line number of the source line within that file.

Table 4–3 lists the four levels of diagnostics that the UIL compiler can issue, in ascending order of severity.

Table 4–3 Levels of Diagnostic Messages

Severity Level	Code	Compilation Status
Informational	I	Accompanies other diagnostics that should be investigated.
Warning	W	Compilation continues. Check that the compiler did what you expected.
Error	E	Compilation continues. No UID file is generated.
Fatal (severe)	F	Compilation terminates immediately.

4.4 Interpreting the Compiler Listing

The listing produced by the compiler contains the following information:

- A title giving miscellaneous information about the compilation
- Source lines of the input file
- Source lines of any include files
- Diagnostics issued by the compiler
- Summary of the diagnostics issued
- Summary of the files read

Example 4–1 shows the contents of a sample listing file.

Example 4–1 Sample UIL Compiler Listing File

```
VMS UIL Compiler V2.0-000       19-APR-1999 15:47:34.31          Page 1
Module: EXAMPLE          Version: X-2

    1 (0)       MODULE example VERSION = 'X-2'
    2 (0)
    3 (0)           INCLUDE FILE 'colors.uil';
    1 (1)
    2 (1)         VALUE red: COLOR( 91 );  VALUE green: COLOR( 92 );
                                  1                              2
%UIL-E-WRONG_TYPE (1) found integer value when expecting string value
%UIL-E-WRONG_TYPE (2) found integer value when expecting string value

    3 (1)         VALUE blue: COLOR( 'Blue' );
    4 (1)
    4 (0)
    5 (0)           OBJECT primary_window:
    6 (0)               MAIN_WINDOW
    7 (0)                   { ARGUMENTS
    8 (0)                         { FOREGROUND_COLOR = pink;
                                                       1
%UIL-E-UNDEFINED (1) value PINK must be defined before this reference
%UIL-E-OBJ_TYPE (1) found error value when expecting color value
```

(continued on next page)

Example 4–1 (Cont.) Sample UIL Compiler Listing File

```
    9 (0)                          BACKGROUND_COLOR = blue; }};
                                                                1
%UIL-E-SYNTAX (1) unexpected RIGHT_BRACE token seen - parsing resumes after ";"

   10 (0)                      };
   11 (0)       END MODULE;
%UIL-I-SUMMARY (0) errors: 5  warnings: 0  informationals: 0

    file (0)    DISK$:[PROJECT.UIL.WORK]A.UIL;1
    file (1)    DISK$:[PROJECT.UIL.WORK]COLOR.UIL;2
```

4.4.1 Listing Title

Each new page of the listing starts with a title. The following example shows a sample UIL listing title:

```
VMS UIL Compiler V2.0-000       19-APR-1999 15:47:34.31         Page 1
Module: EXAMPLE          Version: X-2
```

The first line of the title identifies the compiler by name, the version of the compiler used for the compilation, and the time the compilation started. The version of the compiler, in this case V2.0–000, is important information to include with each Software Performance Report (SPR) you submit concerning the UIL compiler. The version precisely identifies which variant of the compiler you are using.

The second line of the title lists the module name for the UIL specification and the version of that module (if provided in a version clause on the module declaration).

4.4.2 Source Line

Each source line printed on a UIL compiler listing begins with two numbers. These numbers are not part of the source line; the UIL compiler includes these numbers for your convenience. The following example of a line from a compiler listing shows these numbers:

```
1 (0)         MODULE example VERSION = 'X-2'
```

The number in parentheses designates from which file the source line was read. By looking at the file summary at the end of the listing, you see that file (0) is DISK$:[PROJECT.UIL.WORK]A.UIL;1. The first number on the printed line is the number of the source line in the source file.

If a source line contains any control characters other than tabs, they are replaced in the listing by question marks (?).

4.4.3 Diagnostics

Diagnostics for a particular source line follow that line in the listing. Example 4–2 shows some sample source lines that contain errors and the diagnostics associated with those errors.

Example 4–2 Diagnostics on a UIL Compiler Listing

```
    2 (1)           VALUE red: COLOR( 91 );  VALUE green: COLOR( 92 );
                                          1                          2
%UIL-E-WRONG_TYPE (1) found integer value when expecting string value
%UIL-E-WRONG_TYPE (2) found integer value when expecting string value

    8 (0)                           { FOREGROUND_COLOR = pink;
                                                         1
%UIL-E-UNDEFINED (1) value PINK must be defined before this reference
%UIL-E-OBJ_TYPE (1) found error value when expecting color value
```

The line following the source line points to the position in the source line where each of the diagnostics occurred. You can determine the position of each diagnostic by looking at the number in parentheses that follows the diagnostic abbreviation. For example, diagnostic (1) for source line 2 is at position 1 and diagnostic (2) for the same source line is at position 2. The diagnostics for source line 8 both occur at position (1).

If a diagnostic has no associated position on the line, the position is given as (0). If a diagnostic is not associated with a source line, it appears following the last source line. The summary diagnostic, which tallies the number of diagnostics of each severity level, is an example of a diagnostic with no source line association.

4.4.4 Summaries

The UIL compiler listing contains two summaries. Example 4–3 shows the summaries on a sample UIL compiler listing.

Example 4–3 Summaries on a UIL Compiler Listing

```
❶ %UIL-I-SUMMARY (0) errors: 5  warnings: 0  informationals: 0

❷       file (0)     DISK$:[PROJECT.UIL.WORK]A.UIL;1
           file (1)     DISK$:[PROJECT.UIL.WORK]COLOR.UIL;2
```

❶ The first summary is an informational diagnostic that tallies the number of error, warning, and informational diagnostics.

❷ The second summary lists the files that contributed to the UIL specification. This list is useful in determining from which file a source line in the listing was read. For example, a source line preceded by the sequence 532 (3) in the listing was read from line 532 of file (3) in the summary list.

A UIL Diagnostic Messages

In this appendix, the following strings are used to represent data that varies in the actual message you receive from the UIL compiler:

String	Data Represented
%c	Character
%d	Decimal number
%s	String

Messages are listed alphabetically by IDENT code.

ARG_COUNT, procedure %s was previously declared with %d arguments

Severity: Error

Explanation: The declaration of the marked procedure specified a different number of arguments than are present in this procedure reference.

User Action: Check that you are calling the correct function. If you intend to call the procedure with a varying number of arguments, omit the argument list in the procedure declaration.

ARG_TYPE, found %s value - procedure %s argument must be %s value

Severity: Error

Explanation: The declaration of the marked procedure specified a different type of argument than is present in this procedure reference.

User Action: Check that you are passing the correct argument to the correct function. If you intend to call the procedure with varying argument types, declare the procedure specifying ANY for the type of the argument.

BACKSLASH_IGNORED, unknown escape sequence "\%c" - ignored

Severity: Error

Explanation: A backslash (\ ) was followed by an unknown escape character. The backslash is the escape character in UIL. A selected set of single characters can follow a backslash, such as \n for newline or \\ to insert a backslash. The character following the backslash was not one of the selected set.

User Action: If you want to add a backslash, use \\. See Chapter 2 for a description of the supported escape sequences.

BUG_CHECK, Internal error: %s

Severity: Severe

Explanation: The compiler diagnosed an internal error.

User Action: Submit a Software Performance Report (SPR).

CIRCULAR_DEF, widget %s is part of a circular definition

Severity: Error

Explanation: The indicated object is referenced as a descendant of itself, either within its own definition or within the definition of one of the objects in the widget hierarchy that it controls.

User Action: Change the definition of the indicated object so that it is not a descendant of itself.

CONTROL_CHAR, unprintable character \%d\ ignored

Severity: Error

Explanation: The compiler encountered an illegal control character in the UIL specification file. The decimal value of the character is given between the backslash (\ ) characters.

User Action: Replace the character with the sequence specified in the message (for example, \3\ if the control character's internal value is 3). UIL provides several built-in control characters, such as \n and \r for newline and carriage return. See Chapter 2 for a complete list of supported escape sequences.

CREATE_PROC, creation procedure is not supported by the %s widget

Severity: Error

Explanation: You specified a creation procedure for an XUI Toolkit widget. You can specify a creation procedure only for a user-defined widget.

User Action: Remove the procedure clause following the widget type.

CREATE_PROC_INV, creation procedure is not allowed in a %s widget reference

Severity: Error

Explanation: You specified a creation procedure when referring to a widget. You can specify a creation procedure only when you declare the widget.

User Action: Remove the procedure clause following the widget type.

CREATE_PROC_REQ, creation procedure is required in a %s widget declaration

Severity: Error

Explanation: When defining a user-defined widget, you must specify the name of the low-level creation routine for creating an instance of this widget.

User Action: Insert a procedure clause following the widget type in the widget declaration. You also need to declare the creation procedure using a procedure declaration. For example:

```
procedure my_list_box_creation_proc();

object list_box1:
    user_defined procedure my_list_box_creation_proc()
      { arguments ... };
```

CTX_REQ, context requires a %s - %s was specified

Severity: Error

Explanation: At the point marked in the specification, one type of object (such as a widget) is required and your specification supplied a different type of object (such as a value).

User Action: Check for misspelling or that you have referred to the intended object.

D_VALUE_TOO_LARGE, value %s is too large for context buffer

Severity: Severe

Explanation: The compiler encountered a table that is too large for DRM context buffers.

User Action: Break the table into two or more sections.

DUP_LETTER, color letter used for prior color in this table

Severity: Error

Explanation: Each of the letters used to represent a color in a color table must be unique. If not, that letter in an icon would represent more than one color; each pixel can have only one color associated with it at a time. The letter marked has been assigned to more than one color.

User Action: Choose which color the letter is to represent and remove or assign a new character to any duplicates.

DUP_LIST, %s %s already specified for this %s %s

Severity: Error

Explanation: A widget or gadget declaration can have at most one arguments list, one callbacks list, and one controls list.

User Action: If you want to specify multiple lists of arguments, controls, and callbacks, you can do so within one list. For example:

```
arguments { arguments_list1; arguments_list2; };
```

GADGET_NOT_SUP, %s gadget is not supported - %s widget will be used instead

Severity: Warning

Explanation: The indicated object type does not support a gadget variant; only a widget variant is supported for this object type. The UIL compiler ignores the gadget indication, and creates widgets of this object type.

User Action: Specify that this object type is a widget instead of a gadget.

ICON_LETTER, row %d, column %d: letter \"%c"\ not in color table

Severity: Error

Explanation: You have specified a color to be used in an icon that is not in that icon's color table. The invalid color is identified in the message by displaying the letter used to represent that color between the backslashes (\\) . This letter was not defined in the specified color table.

User Action: Either add the color to the icon's color table or use a character representing a color in the color table. The default color table defines " " as background and "*" as foreground.

ICON_WIDTH, row %d must have same width as row 1

Severity: Error

Explanation: The icons supported by UIL are rectangular (that is, x pixels wide by y pixels high). As a result, each of the strings used to represent a row of pixels in an icon must have the same length. The specified row does not have the same length as the first row.

User Action: Make all the strings in the icon function the same length.

INV_MODULE, invalid module structure - check UIL module syntax

Severity: Error

Explanation: The structure of the UIL module is incorrect.

User Action: If there are any syntax errors reported, fix them and recompile. For example, if the error occurs before the first object declaration (that is, before your value and object declarations), check the syntax of the module header for unwanted semicolons (;) after the module clauses. If the error occurs at the end of the module, check that the module concludes with the clause "end module;".

LIST_ITEM, %s item not allowed in %s %s

Severity: Error

Explanation: The indicated list item is not of the type required by the list. Arguments lists must contain argument entries, callbacks lists must contain callback entries, and controls lists must contain control entries.

User Action: Check the syntax for the type of list entry that is required in this context and change the indicated list item.

LISTING_OPEN, error opening listing file: %s

Severity: Severe

Explanation: The compiler could not create the listing file noted in the message.

User Action: Check that you have write access to the directory you specified to hold the listing file.

LISTING_WRITE, error writing to listing file: %s

Severity: Severe

Explanation: The compiler could not write a line into the listing file noted in the message.

User Action: Check to see that there is adequate space on the disk specified to hold the listing file.

NAME_TOO_LONG, name exceeds 31 characters - truncated to: %s

Severity: Error

Explanation: The UIL compiler encountered a name longer than 31 characters. The compiler truncated the name to the leftmost 31 characters.

User Action: Shorten the name in the UIL module source.

NAMES, place names clause before other module clauses

Severity: Error

Explanation: The case-sensitivity clause, if specified, must be the first clause following the module's name. You have inserted another module clause before this clause.

User Action: Reorder the module clauses so that the case-sensitivity clause is first. (It is acceptable to place the version clause ahead of the case-sensitivity clause; this is the only exception.)

NEVER_DEF, %s %s was never defined

Severity: Error

Explanation: Certain UIL objects such as gadgets and widgets can be referred to before they are defined. The marked object is such an object, however, the compiler never found the object's declaration.

User Action: Check for misspelling. If the module is case sensitive, the spellings of names in declarations and in references must match exactly.

NO_UID, no UID file was produced

Severity: Informational

Explanation: If the compiler reported error or severe diagnostics (that is, any of the diagnostic abbreviations starting with %UIL-E or %UIL-F), a UID file is not created. This diagnostic informs you that the compiler did not produce a UID file.

User Action: Fix the problems reported by the compiler.

NON_PVT, value used in this context must be private

Severity: Error

Explanation: A private value is one that is not imported or exported. In the context marked by the message, only a private value is legal. This message may be issued when you have defined one value in terms of another, or when you have defined arguments of functions. In general, a value must be private when the compiler must know the value at compilation time. Exported values are disallowed in these contexts, even though a value is present, because those values could be overridden at run time.

User Action: Change the value to be private.

NOT_IMPL, %s is not implemented yet

Severity: Error

Explanation: You are using a feature of UIL that has not been implemented.

User Action: Try an alternate technique.

NULL, a NULL character in a string is not supported

Severity: Warning

Explanation: You have created a string that has an embedded null character. Strings are represented in a UID file and in many XUI Toolkit data structures as null-terminated strings. So, although the embedded nulls will be placed in the UID file, XUI Toolkit routines may interpret an embedded null as the terminator for the string.

User Action: Use embedded nulls very carefully.

OBJ_TYPE, found %s %s when expecting %s %s

Severity: Error

Explanation: Most arguments take values of a specific type. The value specified is not correct for this argument.

User Action: The message indicates the expected type of argument. Check that you have specified the intended value and that you specified the correct argument.

OPERAND_TYPE, %s type is not valid for %s

Severity: Error

Explanation: The indicated operand is not of a type that is supported by this operator.

User Action: Check the definition of the operator and make sure the type of the operand you specify is supported by the operator.

OUT_OF_MEMORY, compiler ran out of virtual memory

Severity: Severe

Explanation: The compiler ran out of virtual memory.

User Action: Reduce the size of your application.

OUT_RANGE, value of %s is out of range %s

Severity: Error

Explanation: The value specified is outside the legal range of its type.

User Action: Change the UIL module source.

PREV_ERROR, compilation terminated - fix previous errors

Severity: Severe

Explanation: Errors encountered during the compilation have caused the compiler to abort.

User Action: Fix the errors already diagnosed by the compiler and recompile.

PREVIOUS_DEF, name %s previously defined as %s

Severity: Error

Explanation: The name marked by the message was used in a previous declaration. UIL requires that the names of all objects declared within a module be unique.

User Action: Check for a misspelling. If the module is case sensitive, the case of names in declarations and in references must match exactly.

SINGLE_LETTER, color letter string must be a single character

Severity: Error

Explanation: The string associated with each color in a color table must hold exactly one character. You have specified a string with either fewer or more characters.

User Action: Use a single character to represent each color in a color table.

SINGLE_OCCUR, %s %s supports only a single %s %s

Severity: Warning

Explanation: You have specified a particular clause more than once in a context where that clause can occur only once. For example, the version clause in the module can occur only once.

User Action: Choose the correct clause and delete the others.

SRC_LIMIT, too many source files open: %s

Severity: Severe

Explanation: The compiler has a fixed limit for the number of source and include files that it can process. This number is reported in the message.

User Action: Use fewer include files.

SRC_NULL_CHAR, source line contains a null character

Severity: Error

Explanation: The specified source line contains a null character. The compiler ignores any text following the null character.

User Action: Replace each null character with the escape sequence \0\.

SRC_OPEN, error opening source file: %s

Severity: Severe

Explanation: The compiler could not open the UIL specification file listed in the message.

User Action: Check that the file listed in the message is the one you want to compile, that it exists, and that you have read access to the file. If you are using a large number of include files, you may have exceeded your quota for open files.

SRC_READ, error reading next line of source file: %s

Severity: Severe

Explanation: The compiler could not read a line of the UIL specification file listed in the message.

User Action: In the listing file, this message should appear following the last line the compiler read successfully. First check that the file you are compiling is a UIL specification file. If it is, the file mostly likely contains corrupted records.

SRC_TRUNCATE, line truncated at %d characters

Severity: Error

Explanation: The compiler encountered a source line greater than 132 characters. Characters beyond the 132-character limit were ignored.

User Action: Break each source line longer than 132 characters into several source lines. Long string literals can be created using the concatenation operator.

SUBMIT_SPR, internal error - submit an SPR

Severity: Severe

Explanation: The compiler diagnosed an internal error.

User Action: Get a listing and look where the error is being issued. Try fixing any faulty syntax in this area. If you are unable to prevent this error, submit a Software Performance Report (SPR).

SUBNOTALL, Subqualifier not allowed with negated qualifier

Severity: Severe

Explanation: Subqualifiers are not allowed for the negative form of this qualifier. For example, /NOWARNINGS is correct but /NOWARNINGS=NOINFORMATIONALS is not allowed.

User Action: Use the positive form of the qualifier. For example, /WARNINGS=NOINFORMATIONALS.

SUMMARY, errors: %d warnings: %d informationals: %d

Severity: Informational

Explanation: This message lists a summary of the diagnostics issued by the compiler, and appears only when diagnostics have been issued.

User Action: Fix the problems reported. You can use the /WARNINGS qualifier to suppress informational and warning diagnostics that you have determined to be harmless.

SUPERSEDE, this %s %s supersedes a previous definition in this %s %s

Severity: Informational

Explanation: An arguments or callbacks list has either a duplicate argument or a duplicate reason.

User Action: This is not necessarily an error. The compiler is alerting you to make sure that you intended to override an prior argument's value. This informational message can be suppressed using the /WARNINGS qualifier.

SYNTAX, unexpected %s token seen - parsing resumes after \"%c"\

Severity: Error

Explanation: At the point marked in the module, the compiler found a construct such as a punctuation mark, name, or keyword when it was expecting a different construct. The compiler continued analyzing the module at the next occurrence of the construct stated in the message.

User Action: Check the syntax of your UIL module at the point marked by the compiler. If the module specifies case-sensitive names, check that your keywords are in lowercase characters.

TOO_MANY, too many %ss in %s, limit is %d

Severity: Error

Explanation: You exceeded a compiler limit, such as the number of fonts in a font table or the number of strings in a translation table. The message indicates the limit imposed by the compiler.

User Action: Restructure your UIL module.

UID_OPEN, error opening UID file: %s

Severity: Severe

Explanation: The compiler could not create the UID file noted in the message. A UID file holds the compiled user interface specification.

User Action: Check that you have write access to the directory you specified to hold the UID file. If you have a large number of source and include files, check that you have not exceeded your open file quota.

UNDEFINED, %s %s must be defined before this reference

Severity: Error

Explanation: The object pointed to in the message was either never defined or not defined prior to this point in the module. The compiler requires the object to be defined before you refer to the object.

User Action: Check for a misspelling of the object's name, a missing declaration for the object, or a declaration of the object after its first reference. If names in the module are case sensitive, the case of the name in the declaration and in the reference must match exactly.

UNKNOWN_CHARSET, unknown character set

Severity: Error

Explanation: The message is pointing to a context where a character set name is required. You have not specified the name of a character set in that context.

User Action: Check for misspelling. A list of the supported character sets is given in Chapter 2. If you specified case-sensitive names in the module, check that the character set name is in lowercase characters.

UNKNOWN_SEQ, unknown sequence \"%s"\ ignored

Severity: Error

Explanation: The compiler detected a sequence of printable characters it did not understand. The compiler omitted the sequence of characters listed between the quotation marks (" ").

User Action: Fix the UIL module source.

UNSUPPORTED, the %s %s is not supported for the %s %s

Severity: Warning

Explanation: Each widget or gadget supports a specific set of arguments, reasons, and children. The particular argument, reason, or child you specified is not supported for this widget or gadget.

User Action: See the UIL built-in tables in Appendix B for the arguments, reasons, and children supported for each object. If a low-level creation routine accepts an argument that UIL rejects, this does not necessarily indicate that the UIL compiler is in error. Low-level routines ignore arguments that they do not support, without notifying you that the argument is being ignored.

UNTERM_SEQ, %s not terminated %s

Severity: Error

Explanation: The compiler detected a sequence that was not properly terminated, such as a string literal without the closing quotation mark.

User Action: Insert the proper termination characters.

WRONG_TYPE, found %s value when expecting %s value

Severity: Error

Explanation: The indicated value is not of the specific type required by UIL in this context.

User Action: Check the definition of the function or clause.

B UIL Built-In Tables

This appendix contains a listing of the UIL built-in tables used during compilation to check that your UIL specification is consistent with the XUI Toolkit.

For each object in the XUI Toolkit, this appendix contains a table that lists the reasons, controls (children), and arguments (widget attributes) supported by UIL for the object. In addition, the data type and default value of each supported argument are listed. The tables are arranged alphabetically by object name.

UIL has its own generic data types for arguments. These generic data types map to VAX or MIT C binding data types. UIL forces you to specify argument values of the correct data type, and is more structured than the Toolkit in this regard. When you use UIL to specify an interface, you must use UIL data types as indicated in the UIL built-in tables in this appendix.

The default values have been included in this appendix to give an indication of the behavior of the object when you do not specify particular arguments in the UIL module. For example, if you do not specify the border width for an object, the object is displayed with a border that is 1 pixel wide. These defaults implement the appearance of objects described in the *XUI Style Guide*.

For a complete description of the arguments, refer to the low-level creation routine for the object in the *VMS DECwindows Toolkit Routines Reference Manual*. The common arguments, or common attributes, supported by all objects are described in the introduction to Chapter 8, which contains the low-level creation routine descriptions. Chapter 8 also explains the widget class hierarchy and attribute inheritance, which you need to understand in order to locate the description of arguments (attributes) that the object inherits from its superclasses.

In general, the UIL name for an argument closely resembles the XUI Toolkit name for that argument. (Arguments are called widget-specific attributes in the *VMS DECwindows Toolkit Routines Reference Manual*. Callback attributes in the XUI Toolkit are represented by reasons in UIL.) In those cases where the XUI Toolkit attribute name is a UIL keyword, the UIL name for the argument has been configured to avoid ambiguity; the argument name is appended to the object name (or an abbreviated form of the object name).

For example, some objects have a **label** argument. Because label is also a type of object, the UIL name for this argument is **label_label**. To take another example, the command window widget has a **value** argument. Because VALUE is a UIL keyword, the UIL name for this argument is **command_value**.

Some arguments take a symbolic reference to a widget identifier as their value. These arguments are denoted by the term **object reference** in the Type column for the argument. To specify a symbolic reference to a widget identifier, supply the type and name of an object that is defined in your UIL module. For example:

```
    .
    .
    .
arguments {
    default_button = push_button my_push_button;
};
```

Table B–1 lists the UIL object type for each widget class in the XUI Toolkit. DRM uses the object type to determine the low-level creation routine call to construct. Use the generic name to locate the corresponding high- and low-level widget creation routines in the *VMS DECwindows Toolkit Routines Reference Manual*.

Table B–1 UIL Object Types

Generic Name	UIL Name
Attached dialog box	attached_dialog_box
Caution box	caution_box
Color mix	color_mix
Command window	command_window
Compound string text	compound_str_txt
Dialog box	dialog_box
File selection	file_selection
Help box	help_box
Label widget	label_label
Label gadget	label_label
List box	list_box
Main window	main_window
Menu bar	menu_bar
Message box	message_box
Option menu	option_menu
Popup attached dialog box	popup_attached_db
Popup dialog box	popup_dialog_box
Popup menu	popup_menu
Pulldown menu entry widget	pulldown_entry
Pulldown menu entry gadget	pulldown_entry
Pulldown menu	pulldown_menu
Push button widget	push_button
Push button gadget	push_button

(continued on next page)

Table B–1 (Cont.) UIL Object Types

Generic Name	UIL Name
Radio box	radio_box
Scale	scale
Scroll bar	scroll_bar
Scroll window	scroll_window
Selection	selection
Separator widget	separator
Separator gadget	separator
Simple text	simple_text
Toggle button widget	toggle_button
Toggle button gadget	toggle_button
User defined	user_defined
Window	window
Work area menu	work_area_menu
Work-in-progress box	work_in_progress_box

B.1 Attached Dialog Box

Table B–2 shows the controls, reasons, and arguments supported by the attached dialog box widget.

Table B–2 Attached Dialog Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
attached_dialog_box	create	accelerators	translation_ table	Null
caution_box	destroy	adb_bottom_ attachment	integer	DwtAttachNone
color_mix	focus	adb_bottom_offset	integer	0
command_window	help	adb_bottom_position	integer	0
compound_str_txt	map	adb_bottom_widget	object reference	Null
dialog_box	unmap	adb_left_attachment	integer	DwtAttachNone
file_selection		adb_left_offset	integer	0
help_box		adb_left_position	integer	0
label[1]		adb_left_widget	object reference	Null
list_box		adb_right_attachment	integer	DwtAttachNone
main_window		adb_right_offset	integer	0
menu_bar		adb_right_position	integer	0
message_box		adb_right_widget	object reference	Null
option_menu		adb_top_attachment	integer	DwtAttachNone
popup_attached_db		adb_top_offset	integer	0
popup_dialog_box		adb_top_position	integer	0
popup_menu		adb_top_widget	object reference	Null
pulldown_entry[1]		background_color	color	Default background color
pulldown_menu		background_pixmap	pixmap	Null
push_button[1]		border_color	color	Default foreground color
radio_box		border_pixmap	pixmap	Null
scale		border_width	integer	1 pixel
scroll_bar		cancel_button	object reference	Null
scroll_window		default_button	object reference	Null

[1]Widget and gadget variants are supported.

(continued on next page)

Table B–2 (Cont.) Attached Dialog Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
selection		default_horizontal_ offset	integer	0 pixels
separator[1]		default_position	boolean	False
simple_text		default_vertical_offset	integer	0 pixels
toggle_button[1]		direction_r_to_l	boolean	False
user_defined		font_argument	font_table	Default system font
window		fraction_base	integer	100
work_area_menu		grab_merge_ translations	translation_ table	Default translation table syntax
work_in_progress_box		height	integer	5 pixels or as large as needed to hold children
		mapped_when_ managed	boolean	True
		margin_height	integer	1 pixel
		margin_width	integer	1 pixel
		resizable	boolean	True
		resize	integer	DwtResizeGrowOnly
		resize_mode	integer	DwtResizeGrowOnly
		rubber_positioning	boolean	False
		sensitive	boolean	True
		text_merge_ translations	translation_ table	Null
		translations	translation_ table	Default translation syntax
		units	integer	DwtFontUnits
		user_data	any	Null
		width	integer	5 pixels or as large as needed to hold children
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

[1]Widget and gadget variants are supported.

B.2 Caution Box

Table B–3 shows the controls, reasons, and arguments supported by the caution box widget.

Table B–3 Caution Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	cancel	accelerators	translation_table	Null
	create	auto_unmanage	boolean	True
	destroy	auto_unrealize	boolean	False
	focus	background_color	color	Default background color
	help	background_pixmap	pixmap	Null
	map	border_color	color	Default foreground color
	no	border_pixmap	pixmap	Null
	unmap	border_width	integer	1 pixel
	yes	cancel_label	compound_string	Null
		default_position	boolean	False
		default_pushbutton	integer	DwtYesButton
		direction_r_to_l	boolean	False
		font_argument	font_table	Default system font
		height	integer	5 pixels or as large as needed to hold children
		icon_pixmap	pixmap	Null
		label_alignment	integer	DwtAlignmentCenter
		label_label	compound_string	Object name
		mapped_when_managed	boolean	True
		margin_height	integer	10 pixels
		margin_width	integer	12 pixels
		no_label	compound_string	"No" (DwtSNo)
		no_resize	boolean	No (no window manager resize button)
		resize	integer	DwtResizeShrinkWrap
		resize_mode	integer	DwtResizeShrinkWrap
		second_label	compound_string	Null

(continued on next page)

Table B–3 (Cont.) Caution Box

Controls	Reasons	Arguments		
		Name	**UIL Data Type**	**Default Value**
		second_label_alignment	integer	DwtAlignmentBeginning
		sensitive	boolean	True
		style	integer	DwtModal
		take_focus	boolean	True (modal); False (modeless)
		title	compound_string	Object name
		translations	translation_table	Default translation table syntax
		user_data	any	Null
		width	integer	5 pixels or as large as needed to hold children
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager
		yes_label	compound_string	"Yes" (DwtSYes)

B.3 Color Mix

Table B–4 shows the controls, reasons, and arguments supported by the color mix widget.

Table B–4 Color Mix

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	apply	accelerators	translation_table	Null
	cancel	apply_label	compound_string	"Apply"
	create	auto_unmanage	boolean	True
	destroy	auto_unrealize	boolean	False
	focus	back_blue_value	color	
	help	back_green_value	color	
	map	back_red_value	color	
	ok	background_color	color	Default background color
	unmap	background_pixmap	pixmap	Null
		black_label	compound_string	"Black"
		blue_label	compound_string	"Blue"
		border_color	color	Default foreground color
		border_pixmap	pixmap	Null
		border_width	integer	1 pixel
		cancel_button	object reference	Null
		cancel_label	compound_string	"Cancel"
		child_overlap	boolean	True
		color_model	integer	DwtColorModelHLS
		default_button	object reference	Null
		default_position	boolean	False
		direction_r_to_l	boolean	False
		disp_win_margin	integer	20
		display_col_win_height	integer	80
		display_col_win_width	integer	80
		display_label	compound_string	Null

(continued on next page)

Table B–4 (Cont.) Color Mix

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
		display_window	object reference	Null
		font_argument	font_table	Default system font
		full_label	compound_ string	"Full"
		grab_merge_ translations	translation_ table	Default translation table syntax
		gray_label	compound_ string	"Gray"
		green_label	compound_ string	"Green"
		height	integer	5 pixels or as large as needed to hold children
		hls_label	compound_ string	"HLS"
		hue_label	compound_ string	"Hue:"
		light_label	compound_ string	"Lightness:"
		main_label	compound_ string	Null
		mapped_when_ managed	boolean	True
		margin_height	integer	3 pixels
		margin_width	integer	3 pixels
		match_colors	boolean	
		mixer_label	compound_ string	Null
		mixer_window	object reference	Null
		new_blue_value	color	
		new_disp_window	object reference	Null
		new_green_value	color	
		new_red_value	color	
		ok_label	compound_ string	"Ok"
		option_label	compound_ string	"Color Model: "

(continued on next page)

Table B–4 (Cont.) Color Mix

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
		orig_blue_value	color	
		orig_disp_window	object reference	Null
		orig_green_value	color	
		orig_red_value	color	
		red_label	compound_ string	"Red"
		reset_label	compound_ string	"Reset"
		resize	integer	DwtResizeGrowOnly
		resize_mode	integer	DwtResizeGrowOnly
		rgb_label	compound_ string	"RGB"
		sat_label	compound_ string	"Saturation:"
		sensitive	boolean	True
		set_new_color_proc	any	
		slider_label	compound_ string	"Percentage"
		style	integer	DwtModeless
		text_merge_ translations	translation_ table	Null
		title	compound_ string	Color Mixing
		translations	translation_ table	Default translation table syntax
		units	integer	DwtFontUnits
		user_data	any	Null
		value_label	compound_ string	"Value"

(continued on next page)

Table B–4 (Cont.) Color Mix

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
		white_label	compound_ string	"White"
		width	integer	5 pixels or as large as needed to hold children
		work_window	object reference	Null
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.4 Command Window

Table B–5 shows the controls, reasons, and arguments supported by the command window widget.

Table B–5 Command Window

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	command_entered	accelerators	translation_table	Null
	create	background_color	color	Default background color
	destroy	background_pixmap	pixmap	Null
	focus	border_color	color	Default foreground color
	help	border_pixmap	pixmap	Null
	map	border_width	integer	1 pixel
	unmap	command_value	string	Null
	value_changed	default_position	boolean	True
		direction_r_to_l	boolean	False
		font_argument	font_table	Default system font
		height	integer	51 pixels or as large as needed to show history lines and command line
		history	string	"" (Null string)
		lines	integer	2 lines
		mapped_when_managed	boolean	True
		margin_height	integer	1 pixel
		margin_width	integer	1 pixel
		prompt	compound_string	">"
		sensitive	boolean	True
		text_translation	translation_table	Null
		translations	translation_table	Default translation table syntax
		user_data	any	Null
		width	integer	33 pixels
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.5 Compound String Text Widget

Table B–6 shows the controls, reasons, and arguments supported by the compound string text widget widget.

Table B–6 Compound String Text Widget

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	activate	auto_show_insertion_point	boolean	True
	create	background_color	color	Default background color
	destroy	background_pixmap	pixmap	Null
	focus	bidirectional_cursor	boolean	False
	help	blink_rate	integer	500 milliseconds
	lost_focus	border_color	color	Default foreground color
	no_font	border_pixmap	pixmap	Null
	value_changed	border_width	integer	1 pixel
		cols	integer	1
		compound_text_value	compound_string	Null
		direction_r_to_l	boolean	False
		editable	boolean	True
		font_argument	font_table	Default system font
		foreground_color	color	Default foreground color
		half_border	boolean	True
		height	integer	(rows * font height) + (2 * margin height)
		insertion_point_visible	boolean	True
		insertion_position	integer	0
		mapped_when_managed	boolean	True
		margin_height	integer	2 pixels
		margin_width	integer	2 pixels
		max_length	integer	MAXINT; the largest number that can fit in a C long ((2**31)-1, or 2,147,483,647)
		pending_delete	boolean	True
		resize_height	boolean	True
		resize_width	boolean	True

(continued on next page)

Table B–6 (Cont.) Compound String Text Widget

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
		rows	integer	20
		scroll_horizontal	boolean	False
		scroll_left_side	boolean	False
		scroll_top_side	boolean	False
		scroll_vertical	boolean	False
		sensitive	boolean	True
		top_position	integer	0
		translations	translation_ table	Default translation table syntax
		widget_direction	integer	False
		width	integer	(columns * average character width) + (2 * margin width)
		word_wrap	boolean	False
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.6 Dialog Box

Table B–7 shows the controls, reasons, and arguments supported by the dialog box widget.

Table B–7 Dialog Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
attached_dialog_box	create	accelerators	translation_ table	Null
caution_box	destroy	background_color	color	Default background color
color_mix	focus	background_pixmap	pixmap	Null
command_window	help	border_color	color	Default foreground color
compound_str_txt	map	border_pixmap	pixmap	Null
dialog_box	unmap	border_width	integer	1 pixel
file_selection		cancel_button	object reference	Null
help_box		child_overlap	boolean	True
label[1]		default_button	object reference	Null
list_box		default_position	boolean	False
main_window		direction_r_to_l	boolean	False
menu_bar		font_argument	font_table	Default system font
message_box		grab_merge_ translations	translation_ table	Default translation table syntax
option_menu		height	integer	5 pixels or as large as needed to hold children
popup_attached_db		mapped_when_ managed	boolean	True
popup_dialog_box		margin_height	integer	3 pixels (popups); 1 pixel (all others)
popup_menu		margin_width	integer	3 pixels (popups); 1 pixel (all others)
pulldown_entry[1]		resize	integer	DwtResizeGrowOnly
pulldown_menu		resize_mode	integer	DwtResizeGrowOnly
push_button[1]		sensitive	boolean	True
radio_box		text_merge_ translations	translation_ table	Null
scale		translations	translation_ table	Default translation table syntax
scroll_bar		units	integer	DwtFontUnits

[1]Widget and gadget variants are supported.

(continued on next page)

Table B–7 (Cont.) Dialog Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
scroll_window		user_data	any	Null
selection		width	integer	5 pixels or as large as needed to hold children
separator[1]		x	integer	Determined by geometry manager
simple_text		y	integer	Determined by geometry manager
toggle_button[1]				
user_defined				
window				
work_area_menu				
work_in_progress_box				

[1]Widget and gadget variants are supported.

B.7 File Selection

Table B–8 shows the controls, reasons, and arguments supported by the file selection widget.

Table B–8 File Selection

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
attached_dialog_box	activate	accelerators	translation_ table	Null
caution_box	cancel	apply_label	compound_ string	"Apply"
color_mix	create	auto_unmanage	boolean	True
command_window	destroy	auto_unrealize	boolean	False
compound_str_txt	focus	background_color	color	Default background color
dialog_box	help	background_pixmap	pixmap	Null
file_selection	map	border_color	color	Default foreground color
help_box	no_match	border_pixmap	pixmap	Null
label[1]	unmap	border_width	integer	1 pixel
list_box		cancel_label	compound_ string	"Cancel"
main_window		default_position	boolean	False
menu_bar		dir_mask	compound_ string	*.*
message_box		direction_r_to_l	boolean	False
option_menu		file_search_proc	any	FileSelectionSearch
popup_attached_db		file_selection_value	compound_ string	Null
popup_dialog_box		file_to_extern_proc	any	Null
popup_menu		file_to_intern_proc	any	Null
pulldown_entry[1]		filter_label	compound_ string	"File filter"
pulldown_menu		font_argument	font_table	Default system font
push_button[1]		height	integer	5 pixels or as large as needed to hold children
radio_box		items	string_table	Null
scale		label_label	compound_ string	"Files in"
scroll_bar		list_updated	boolean	False
scroll_window		mapped_when_ managed	boolean	True

[1]Gadget variant is not supported.

(continued on next page)

Table B–8 (Cont.) File Selection

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
selection		margin_height	integer	2 pixels
separator[1]		margin_width	integer	2 pixels
simple_text		mask_to_extern_proc	any	Null
toggle_button[1]		mask_to_intern_proc	any	Null
user_defined		must_match	boolean	False
window		no_resize	boolean	True
work_area_menu		ok_label	compound_ string	"Ok"
work_in_progress_box		resize	integer	Grow only (DwtResizeGrowOnly)
		resize_mode	integer	Grow only (DwtResizeGrowOnly)
		selection_label	compound_ string	"Selection"
		sensitive	boolean	True
		style	integer	Modal (DwtModal)
		take_focus	boolean	True (modal); False (modeless)
		text_cols	integer	30
		title	compound_ string	"Open"
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		visible_items_count	integer	8
		width	integer	5 pixels or as large as needed to hold children
		x	integer	Centered in parent
		y	integer	Centered in parent

[1]Gadget variant is not supported.

B.8 Help Box

Table B–9 shows the controls, reasons, and arguments supported by the help box widget.

Table B–9 Help Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	create	about_label	compound_ string	"About"
	destroy	add_topic_label	compound_ string	"Additional topics:"
	help	application_name	compound_ string	Null
	map	background_color	color	Default background color
	unmap	background_pixmap	pixmap	Null
		bad_frame_message	compound_ string	"Couldn't find frame %s in %s library\n"
		bad_library_message	compound_ string	"Couldn't open library %s \n"
		border_color	color	Default foreground color
		border_pixmap	pixmap	Null
		cache_help_library	boolean	True
		close_label	compound_ string	"Close"
		cols	integer	55
		copy_label	compound_ string	"Copy"
		default_position	boolean	True
		direction_r_to_l	boolean	False
		dismiss_label	compound_ string	"Dismiss"
		edit_label	compound_ string	"Edit"
		error_open_message	compound_ string	"Error opening file %s\n"
		exit_label	compound_ string	"Exit"
		file_label	compound_ string	"File"
		first_topic	compound_ string	Null

(continued on next page)

Table B–9 (Cont.) Help Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
		font_argument	font_table	Default system font
		foreground_color	color	Default foreground color
		glossary_label	compound_ string	"Glossary"
		glossary_topic	compound_ string	Null
		go_back_label	compound_ string	"Go back"
		go_over_label	compound_ string	"Go to Overview"
		go_to_label	compound_ string	"Go To"
		gobacktopic_label	compound_ string	"Go back topic"
		gototopic_label	compound_ string	"Go to topic"
		help_acknowledge_ label	compound_ string	"Acknowledge"
		help_font	font_table	Default Help font"
		help_label	compound_ string	"Help"
		help_message_title	compound_ string	"Message"
		help_message_title_ type	integer	DwtCString
		help_on_help_title	compound_ string	"Using Help"
		helphelp_label	compound_ string	"Overview"
		helpontitle_label	compound_ string	"Help On"
		helptitle_label	compound_ string	"Help"
		history_box_label	compound_ string	"Help Topic History"
		history_label	compound_ string	"History..."
		keyword_label	compound_ string	"Keyword..."

(continued on next page)

Table B–9 (Cont.) Help Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
		keywords_label	compound_ string	"Keyword:"
		library_spec	compound_ string	Null
		library_type	integer	1 (text library)
		mapped_when_ managed	boolean	True
		no_keyword_message	compound_ string	"Couldn't find keyword %s\n"
		no_title_message	compound_ string	"No title to match string %s\n"
		null_library_message	compound_ string	"No library specified\n"
		null_topic_message	compound_ string	"No first topic and overview topic specified\n"
		overview_topic	compound_ string	Null
		rows	integer	20
		save_as_label	compound_ string	"Save As..."
		search_apply_label	compound_ string	"Apply"
		search_keyword_box_ label	compound_ string	"Search Topic Keywords"
		search_label	compound_ string	"Search"
		search_title_box_label	compound_ string	"Search Topic Titles"
		select_all_label	compound_ string	"Select All"
		sensitive	boolean	True
		title	compound_ string	"Help"
		title_label	compound_ string	"Title..."
		titles_label	compound_ string	"Title:"
		topic_titles_label	compound_ string	"Topic Titles:"

(continued on next page)

Table B–9 (Cont.) Help Box

Controls	Reasons	Arguments		
		Name	**UIL Data Type**	**Default Value**
		translations	translation_table	Default translation table syntax
		user_data	any	Null
		view_label	compound_string	"View"
		visit_glossary_label	compound_string	"Visit Glossary"
		visit_label	compound_string	"Visit"
		visittopic_label	compound_string	"Visit topic"
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.9 Label Widget

Table B–10 shows the controls, reasons, and arguments supported by the label widget.

Table B–10 Label Widget

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	create	alignment	integer	DwtAlignmentCenter
	destroy	background_color	color	Default background color
	help	background_pixmap	pixmap	Null
		border_color	color	Default foreground color
		border_pixmap	pixmap	Null
		border_width	integer	0 pixels
		conform_to_text	boolean	True (if widget created without width and height); False (if widget created with width and height)
		direction_r_to_l	boolean	False
		font_argument	font_table	Default system font
		foreground_color	color	Default foreground color
		height[1]	integer	0 pixels
		highlight_color	color	Default foreground color
		highlight_pixmap	pixmap	Null
		label_label	compound_ string	Object name
		label_label_type	integer	DwtCString
		label_pixmap	pixmap	Null
		mapped_when_ managed	boolean	True
		margin_bottom	integer	0 pixels
		margin_height	integer	2 pixels (text); 0 pixels (pixmap)
		margin_left	integer	0 pixels
		margin_right	integer	0 pixels
		margin_top	integer	0 pixels
		margin_width	integer	2 pixels (text); 0 pixels (pixmap)

[1] For the widget variant, the default value for the height and width arguments depends on the setting of the conform_to_text argument. If conform_to_text is false, height and width are 0 pixels. If conform_to_text is true, height and width are as large as needed to hold the label text.

(continued on next page)

Table B–10 (Cont.) Label Widget

Controls	Reasons	Arguments		
		Name	**UIL Data Type**	**Default Value**
		sensitive	boolean	True
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		widget_direction	integer	False
		width[1]	integer	0 pixels
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

[1] For the widget variant, the default value for the height and width arguments depends on the setting of the conform_to_text argument. If conform_to_text is false, height and width are 0 pixels. If conform_to_text is true, height and width are as large as needed to hold the label text.

B.10 Label Gadget

Table B–11 shows the controls, reasons, and arguments supported by the label gadget.

Table B–11 Label Gadget

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	create	alignment	integer	DwtAlignmentCenter
	destroy	border_width	integer	0 pixels
	help	direction_r_to_l	boolean	False
		height	integer	0 pixels
		label_label	compound_ string	Object name
		sensitive	boolean	True
		widget_direction	integer	False
		width	integer	0 pixels
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.11 List Box

Table B–12 shows the controls, reasons, and arguments supported by the list box widget.

Table B–12 List Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	create	background_color	color	Default background color
	destroy	background_pixmap	pixmap	Null
	extend	border_color	color	Default foreground color
	extend_ confirm	border_pixmap	pixmap	Null
	help	border_width	integer	1 pixel
	single	create_horizontal_ scroll_bar	boolean	Null
	single_ confirm	direction_r_to_l	boolean	False
		font_argument	font_table	Default system font
		height	integer	As large as needed to hold number of items specified by visible_items_count without exceeding parent widget size
		items	string_table	Null
		mapped_when_ managed	boolean	True
		margin_height	integer	4 pixels
		margin_width	integer	10 pixels
		resize[1]	integer	DwtResizeGrowOnly
		resize_mode[1]	integer	DwtResizeGrowOnly
		selected_items	string_table	Null
		sensitive	boolean	True
		single_selection	boolean	True
		spacing	integer	1 pixel
		translations	translation_ table	Default translation table syntax

[1] The resize argument for the list_box object does not support the value DwtResizeShrinkwrap. You can specify the value TRUE for DwtResizeShrinkwrap or the value FALSE for DwtResizeFixed.

(continued on next page)

Table B–12 (Cont.) List Box

Controls	Reasons	Arguments		
		Name	**UIL Data Type**	**Default Value**
		user_data	any	Null
		visible_items_count	integer	1
		width	integer	As long as needed to hold longest item without exceeding parent widget size
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.12 Main Window

Table B–13 shows the controls, reasons, and arguments supported by the main window widget.

Table B–13 Main Window

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
attached_dialog_box	create	accept_focus	boolean	False
caution_box	destroy	background_color	color	Default background color
color_mix	focus	background_pixmap	pixmap	Null
command_window	help	border_color	color	Default foreground color
compound_str_txt		border_pixmap	pixmap	Null
dialog_box		border_width	integer	1 pixel
file_selection		direction_r_to_l	boolean	False
help_box		foreground_color	color	Default foreground color
label[1]		height	integer	5 pixels
list_box		main_command_window	object reference	Null
main_window		main_horizontal_scroll_bar	object reference	Null
menu_bar		main_menu_bar	object reference	Null
message_box		main_vertical_scroll_bar	object reference	Null
option_menu		main_work_window	object reference	Null
popup_attached_db		mapped_when_managed	boolean	True
popup_dialog_box		sensitive	boolean	True
popup_menu		translations	translation_table	Default translation table syntax
pulldown_entry[1]		user_data	any	Null
pulldown_menu		width	integer	5 pixels
push_button[1]		x	integer	Determined by geometry manager
radio_box		y	integer	Determined by geometry manager
scale				
scroll_bar				
scroll_window				

[1]Gadget variant is not supported.

(continued on next page)

Table B–13 (Cont.) Main Window

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
selection				
separator[1]				
simple_text				
toggle_button[1]				
user_defined				
window				
work_area_menu				
work_in_progress_box				

[1]Gadget variant is not supported.

B.13 Menu Bar

Table B–14 shows the controls, reasons, and arguments supported by the menu bar widget.

Table B–14 Menu Bar

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
label[1]	create	adjust_margin	boolean	True
pulldown_entry[1]	destroy	background_color	color	Default background color
push_button[1]	entry	background_pixmap	pixmap	Null
separator[1]	help	border_color	color	Default foreground color
toggle_button[1]	map	border_pixmap	pixmap	Null
user_defined	unmap	border_width	integer	1 pixel
		change_vis_atts	boolean	True
		direction_r_to_l	boolean	False
		entry_alignment	integer	DwtAlignmentBeginning
		entry_border_width	integer	0
		font_argument	font_table	Default system font
		height	integer	16 pixels or the number of lines needed to display all entries
		mapped_when_managed	boolean	True
		margin_height	integer	3 pixels (vertical orientation); 0 pixels (horizontal orientation)
		margin_width	integer	3 pixels
		menu_alignment	boolean	True
		menu_extend_last_row	boolean	True
		menu_help_widget	object reference	Null
		menu_history	object reference	Null
		menu_num_columns	integer	1
		menu_packing	integer	DwtmenuPackingColumn (radio boxes); DwtmenuPackingTight (all others)
		menu_radio	boolean	True (radio boxes); False (all others)

[1]Widget and gadget variants are supported.

(continued on next page)

Table B–14 (Cont.) Menu Bar

Controls	Reasons	Arguments		
		Name	**UIL Data Type**	**Default Value**
		orientation	integer	DwtOrientationVertical
		radio_always_one	boolean	True
		sensitive	boolean	True
		spacing	integer	1 pixel (vertical orientation); 0 pixels (horizontal orientation)
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		width	integer	16 pixels or the width of parent widget
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.14 Message Box

Table B–15 shows the controls, reasons, and arguments supported by the message box widget.

Table B–15 Message Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	create	accelerators	translation_ table	Null
	destroy	auto_unmanage	boolean	True
	focus	auto_unrealize	boolean	False
	help	background_color	color	Default background color
	map	background_pixmap	pixmap	Null
	unmap	border_color	color	Default foreground color
	yes	border_pixmap	pixmap	Null
		border_width	integer	1 pixel
		default_position	boolean	False
		direction_r_to_l	boolean	False
		font_argument	font_table	Default system font
		height	integer	5 pixels or as large as needed to hold children
		icon_pixmap	pixmap	Null
		label_alignment	integer	DwtAlignmentCenter
		label_label	compound_ string	Object name
		mapped_when_ managed	boolean	True
		margin_height	integer	10 pixels
		margin_width	integer	12 pixels
		no_resize	boolean	True
		ok_label	compound_ string	"Acknowledged"
		resize	integer	Shrink-wrap (DwtResizeShrinkWrap)
		resize_mode	integer	Shrink-wrap (DwtResizeShrinkWrap)
		second_label	compound_ string	Null
		second_label_ alignment	integer	DwtAlignmentBeginning

(continued on next page)

Table B–15 (Cont.) Message Box

Controls	Reasons	Arguments		
		Name	**UIL Data Type**	**Default Value**
		sensitive	boolean	True
		style	integer	Modal (DwtModal)
		take_focus	boolean	False
		title	compound_ string	Object name
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		width	integer	5 pixels or as large as needed to hold children
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.15 Option Menu

Table B–16 shows the controls, reasons, and arguments supported by the option menu widget.

Table B–16 Option Menu

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
pulldown_menu	create	background_color	color	Default background color
user_defined	destroy	background_pixmap	pixmap	Null
	entry	border_color	color	Default foreground color
	help	border_pixmap	pixmap	Null
	map	border_width	integer	0 pixels
	unmap	change_vis_atts	boolean	True
		direction_r_to_l	boolean	False
		entry_alignment	integer	DwtAlignmentBeginning
		entry_border_width	integer	0
		font_argument	font_table	Default system font
		height	integer	16 pixels or as large as needed to hold children
		label_label	compound_ string	Object name
		mapped_when_ managed	boolean	True
		margin_height	integer	3 pixels
		margin_width	integer	3 pixels
		menu_alignment	boolean	True
		menu_history	object reference	Null
		sensitive	boolean	True
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		width	integer	16 pixels or as large as needed to hold children
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.16 Popup Attached Dialog Box

Table B–17 shows the controls, reasons, and arguments supported by the popup attached dialog box widget.

Table B–17 Popup Attached Dialog Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
attached_dialog_box	create	accelerators	translation_ table	Null
caution_box	destroy	adb_bottom_ attachment	integer	DwtAttachNone
color_mix	focus	adb_bottom_offset	integer	0
command_window	help	adb_bottom_position	integer	0
compound_str_txt	map	adb_bottom_widget	object reference	Null
dialog_box	unmap	adb_left_attachment	integer	DwtAttachNone
file_selection		adb_left_offset	integer	0
help_box		adb_left_position	integer	0
label[1]		adb_left_widget	object reference	Null
list_box		adb_right_attachment	integer	DwtAttachNone
main_window		adb_right_offset	integer	0
menu_bar		adb_right_position	integer	0
message_box		adb_right_widget	object reference	Null
option_menu		adb_top_attachment	integer	DwtAttachNone
popup_attached_db		adb_top_offset	integer	0
popup_dialog_box		adb_top_position	integer	0
popup_menu		adb_top_widget	object reference	Null
pulldown_entry[1]		auto_unmanage	boolean	True
pulldown_menu		auto_unrealize	boolean	False
push_button[1]		background_color	color	Default background color
radio_box		background_pixmap	pixmap	Null
scale		border_color	color	Default foreground color
scroll_bar		border_pixmap	pixmap	Null
scroll_window		border_width	integer	1 pixel
selection		cancel_button	object reference	Null

[1] Widget and gadget variants are supported.

(continued on next page)

Table B–17 (Cont.) Popup Attached Dialog Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
separator[1]		default_button	object reference	Null
simple_text		default_horizontal_ offset	integer	0 pixels
toggle_button[1]		default_position	boolean	False
user_defined		default_vertical_offset	integer	0 pixels
window		direction_r_to_l	boolean	False
work_area_menu		font_argument	font_table	Default system font
work_in_progress_box		fraction_base	integer	100
		grab_merge_ translations	translation_ table	Default translation table syntax
		height	integer	5 pixels or as large as needed to hold children
		mapped_when_ managed	boolean	True
		margin_height	integer	3 pixels
		margin_width	integer	3 pixels
		no_resize	boolean	True
		resizable	boolean	True
		resize	integer	DwtResizeGrowOnly
		resize_mode	integer	DwtResizeGrowOnly
		rubber_positioning	boolean	False
		sensitive	boolean	True
		style	integer	DwtModeless
		take_focus	boolean	True (modal); False (modeless)
		text_merge_ translations	translation_ table	Null
		title	compound_ string	Object name
		translations	translation_ table	Default translation table syntax
		units	integer	DwtFontUnits
		user_data	any	Null
		width	integer	5 pixels or as large as needed to hold children

[1]Widget and gadget variants are supported.

(continued on next page)

Table B–17 (Cont.) Popup Attached Dialog Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.17 Popup Dialog Box

Table B–18 shows the controls, reasons, and arguments supported by the popup dialog box widget.

Table B–18 Popup Dialog Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
attached_dialog_box	create	accelerators	translation_ table	Null
caution_box	destroy	auto_unmanage	boolean	True
color_mix	focus	auto_unrealize	boolean	False
command_window	help	background_color	color	Default background color
compound_str_txt	map	background_pixmap	pixmap	Null
dialog_box	unmap	border_color	color	Default foreground color
file_selection		border_pixmap	pixmap	Null
help_box		border_width	integer	1 pixel
label[1]		cancel_button	object reference	Null
list_box		child_overlap	boolean	True
main_window		default_button	object reference	Null
menu_bar		default_position	boolean	False
message_box		direction_r_to_l	boolean	False
option_menu		font_argument	font_table	Default system font
popup_attached_db		grab_merge_ translations	translation_ table	Default translation table syntax
popup_dialog_box		height	integer	5 pixels
popup_menu		mapped_when_ managed	boolean	True
pulldown_entry[1]		margin_height	integer	3 pixels
pulldown_menu		margin_width	integer	3 pixels
push_button[1]		no_resize	boolean	True
radio_box		resize	integer	DwtResizeGrowOnly
scale		resize_mode	integer	DwtResizeGrowOnly
scroll_bar		sensitive	boolean	True
scroll_window		style	integer	DwtModeless
selection		take_focus	boolean	True(modal); False (modeless)

[1] Widget and gadget variants are supported.

(continued on next page)

Table B–18 (Cont.) Popup Dialog Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
separator[1]		text_merge_ translations	translation_ table	Null
simple_text		title	compound_ string	Object name
toggle_button[1]		translations	translation_ table	Default translation table syntax
user_defined		units	integer	DwtFontUnits
window		user_data	any	Null
work_area_menu		width	integer	5 pixels
work_in_progress_box		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

[1]Widget and gadget variants are supported.

B.18 Popup Menu

Table B–19 shows the controls, reasons, and arguments supported by the popup menu widget.

Table B–19 Popup Menu

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
label[1]	create	adjust_margin	boolean	False
pulldown_entry[1]	destroy	background_color	color	Default background color
push_button[1]	entry	background_pixmap	pixmap	Null
separator[1]	help	border_color	color	Default foreground color
toggle_button[1]	map	border_pixmap	pixmap	Null
user_defined	unmap	border_width	integer	0 pixels
		change_vis_atts	boolean	True
		direction_r_to_l	boolean	False
		entry_alignment	integer	DwtAlignmentBeginning
		entry_border_width	integer	0 pixels
		font_argument	font_table	Default system font
		height	integer	16 pixels or as large as needed to hold children
		mapped_when_managed	boolean	True
		margin_height	integer	3 pixels
		margin_width	integer	3 pixels
		menu_alignment	boolean	True
		menu_entry_class	class_rec_name	Null
		menu_extend_last_row	boolean	True
		menu_help_widget	object reference	Null
		menu_history	object reference	Null
		menu_is_homogeneous	boolean	True (radio boxes); False (all others)
		menu_num_columns	integer	1 row or column
		menu_packing	integer	DwtmenuPackingColumn (radio boxes); DwtmenuPackingTight (all others)

[1]Widget and gadget variants are supported.

(continued on next page)

Table B–19 (Cont.) Popup Menu

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
		menu_radio	boolean	True (radio boxes); False (all others)
		orientation	integer	DwtOrientationVertical
		radio_always_one	boolean	True
		sensitive	boolean	True
		spacing	integer	1 pixel
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		width	integer	16 pixels or as large as needed to hold children

B.19 Pulldown Menu Entry Widget

Table B–20 shows the controls, reasons, and arguments supported by the pulldown menu entry widget.

Table B–20 Pulldown Menu Entry Widget

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
pulldown_menu	activate	alignment	integer	DwtAlignmentCenter
user_defined	create	background_color	color	Default background color
	destroy	background_pixmap	pixmap	Null
	help	border_color	color	Default foreground color
	pulling	border_pixmap	pixmap	Null
		border_width	integer	0 pixels
		conform_to_text	boolean	True (if width and height not specified); False (otherwise)
		direction_r_to_l	boolean	False
		font_argument	font_table	Default system font
		foreground_color	color	Default foreground color
		height	integer	0 pixels, or (label or pixmap height) + (2 * margin height)
		highlight_color	color	Default foreground color
		highlight_pixmap	pixmap	Null
		hot_spot_pixmap	pixmap	Null
		label_label	compound_ string	Object name
		label_label_type	integer	DwtCString
		label_pixmap	pixmap	Null
		mapped_when_ managed	boolean	True
		margin_bottom	integer	1 pixel
		margin_height	integer	2 pixels (text); 0 pixels (pixmap)
		margin_left	integer	9 pixels
		margin_right	integer	7 pixels
		margin_top	integer	2 pixels
		margin_width	integer	2 pixels (text); 0 pixels (pixmap)

(continued on next page)

Table B–20 (Cont.) Pulldown Menu Entry Widget

Controls	Reasons	Arguments		
		Name	**UIL Data Type**	**Default Value**
		sensitive	boolean	True
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		width	integer	0 pixels, or (label width + hotspot pixmap width) + (2 * margin width)
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.20 Pulldown Menu Entry Gadget

Table B–21 shows the controls, reasons, and arguments supported by the pulldown menu entry gadget.

Table B–21 Pulldown Menu Entry Gadget

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
pulldown_menu	activate	alignment	integer	DwtAlignmentCenter
user_defined	create	border_width	integer	0 pixels
	destroy	height	integer	0 pixels, or (label or pixmap height) + (2 * margin height)
	help	label_label	compound_ string	Object name
	pulling	sensitive	boolean	True
		width	integer	0 pixels, or (label width + hotspot pixmap width) + (2 * margin width)
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.21 Pulldown Menu

Table B–22 shows the controls, reasons, and arguments supported by the pulldown menu widget.

Table B–22 Pulldown Menu

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
label[1]	create	adjust_margin	boolean	True
pulldown_entry[1]	destroy	background_color	color	Default background color
push_button[1]	entry	background_pixmap	pixmap	Null
separator[1]	help	border_color	color	Default foreground color
toggle_button[1]	map	border_pixmap	pixmap	Null
user_defined	unmap	border_width	integer	0
		change_vis_atts	boolean	True
		direction_r_to_l	boolean	False
		entry_alignment	integer	DwtAlignmentBeginning
		entry_border_width	integer	0
		font_argument	font_table	Default system font
		foreground_color	color	Default foreground color
		height	integer	16 pixels or as large as needed to hold children
		highlight_color	color	Default foreground color
		mapped_when_managed	boolean	True
		margin_height	integer	2 pixels
		margin_width	integer	2 pixels
		menu_alignment	boolean	True
		menu_entry_class	class_rec_name	Null
		menu_extend_last_row	boolean	True
		menu_help_widget	object reference	Null
		menu_history	object reference	Null
		menu_is_homogeneous	boolean	True (radio boxes); False (all others)
		menu_num_columns	integer	1 row or column

[1]Widget and gadget variants are supported.

(continued on next page)

Table B–22 (Cont.) Pulldown Menu

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
		menu_packing	integer	DwtmenuPackingColumn (radio boxes); DwtmenuPackingTight (all others)
		menu_radio	boolean	False
		orientation	integer	DwtOrientationVertical
		radio_always_one	boolean	True
		sensitive	boolean	True
		spacing	integer	0 pixels
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		width	integer	16 pixels or as large as needed to hold children

B.22 Push Button Widget

Table B–23 shows the controls, reasons, and arguments supported by the push button widget.

Table B–23 Push Button Widget

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	activate	accelerator_text	compound_ string	Null
	arm	alignment	integer	DwtAlignmentCenter
	create	background_color	color	Default background color
	destroy	background_pixmap	pixmap	Null
	disarm	border_color	color	Default foreground color
	help	border_highlight	boolean	False
		border_pixmap	pixmap	Null
		border_width[1]	integer	1 pixel
		button_accelerator	string	Null
		conform_to_text	boolean	True (if widget created without width and height); False (if widget created with width and height)
		direction_r_to_l	boolean	False
		fill_highlight	boolean	False
		font_argument	font_table	Default system font
		foreground_color	color	Default foreground color
		height	integer	0 pixels, or (label or pixmap height) + (2 * margin height)
		highlight_color	color	Default foreground color
		highlight_pixmap	pixmap	Null
		insensitive_pixmap	pixmap	Null
		label_label	compound_ string	Object name
		label_label_type	integer	DwtCString
		label_pixmap	pixmap	Null
		mapped_when_ managed	boolean	True
		margin_bottom	integer	1 pixel
		margin_height	integer	4 pixels (text); 0 pixels (pixmap)

[1] For a push button widget, if shadow is true, the default value for border_width is 0 pixels.

(continued on next page)

Table B–23 (Cont.) Push Button Widget

Controls	Reasons	Arguments		
		Name	**UIL Data Type**	**Default Value**
		margin_left	integer	9 pixels
		margin_right	integer	7 pixels
		margin_top	integer	2 pixels
		margin_width	integer	10 pixels (text); 0 pixels (pixmap)
		sensitive	boolean	True
		shadow	boolean	True
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		widget_direction	integer	False
		width	integer	0 pixels, or (label or pixmap width) + (2 * margin width)
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.23 Push Button Gadget

Table B–24 shows the controls, reasons, and arguments supported by the push button gadget.

Table B–24 Push Button Gadget

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	activate	accelerator_text	compound_ string	Null
	create	alignment	integer	DwtAlignmentCenter
	destroy	border_width	integer	1 pixel
	help	button_accelerator	string	Null
		direction_r_to_l	boolean	False
		height	integer	0 pixels, or (label or pixmap height) + (2 * margin height)
		label_label	compound_ string	Object name
		sensitive	boolean	True
		widget_direction	integer	False
		width	integer	0 pixels, or (label or pixmap width) + (2 * margin width)
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.24 Radio Box

Table B–25 shows the controls, reasons, and arguments supported by the radio box widget.

Table B–25 Radio Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
label[1]	create	adjust_margin	boolean	True
pulldown_entry[1]	destroy	background_color	color	Default background color
push_button[1]	entry	background_pixmap	pixmap	Null
separator[1]	help	border_color	color	Default foreground color
toggle_button[1]	map	border_pixmap	pixmap	Null
user_defined	unmap	border_width	integer	0 pixels
		change_vis_atts	boolean	True
		direction_r_to_l	boolean	False
		entry_alignment	integer	DwtAlignmentBeginning
		entry_border_width	integer	0 pixels
		font_argument	font_table	Default system font
		height	integer	16 pixels or as large as needed to hold children
		mapped_when_managed	boolean	True
		margin_height	integer	3 pixels
		margin_width	integer	3 pixels
		menu_alignment	boolean	True
		menu_entry_class	class_rec_name	Null
		menu_extend_last_row	boolean	True
		menu_help_widget	object reference	Null
		menu_history	object reference	Null
		menu_is_homogeneous	boolean	False
		menu_num_columns	integer	1
		menu_packing	integer	DwtmenuPackingTight
		menu_radio	boolean	True
		orientation	integer	DwtOrientationVertical
		radio_always_one	boolean	True

[1]Widget and gadget variants are supported.

(continued on next page)

Table B–25 (Cont.) Radio Box

Controls	Reasons	Arguments		
		Name	**UIL Data Type**	**Default Value**
		sensitive	boolean	True
		spacing	integer	1 pixel
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		width	integer	16 pixels or as large as needed to hold children
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.25 Scale

Table B–26 shows the controls, reasons, and arguments supported by the scale widget.

Table B–26 Scale

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
attached_dialog_box	create	background_color	color	Default background color
caution_box	destroy	background_pixmap	pixmap	Null
color_mix	drag	border_color	color	Default foreground color
command_window	help	border_pixmap	pixmap	Null
compound_str_txt	value_ changed	border_width	integer	0 pixels
dialog_box		decimal_points	integer	0
file_selection		direction_r_to_l	boolean	False
help_box		font_argument	font_table	Default system font
label[1]		foreground_color	color	Default foreground color
list_box		height	integer	Calculated based on scale width or height, label widths, and orientation
main_window		highlight_color	color	Default foreground color
menu_bar		highlight_pixmap	pixmap	Null
message_box		mapped_when_ managed	boolean	True
option_menu		max_value	integer	100
popup_attached_db		min_value	integer	0
popup_dialog_box		orientation	integer	DwtOrientationHorizontal
popup_menu		scale_height	integer	20 pixels
pulldown_entry[1]		scale_value	integer	As close to 0 as possible
pulldown_menu		scale_width	integer	100 pixels
push_button[1]		sensitive	boolean	True
radio_box		show_value	boolean	True
scale		show_value_automatic	boolean	True
scroll_bar		slider_color	color	Default foreground color
scroll_window		slider_pixmap	pixmap	Null
selection		title	compound_ string	Scale name
separator[1]		translations	translation_ table	Default translation table syntax

[1]Gadget variant is not supported.

(continued on next page)

Table B–26 (Cont.) Scale

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
simple_text		user_data	any	Null
toggle_button[1]		width	integer	Calculated based on scale width or height, label widths, and orientation
user_defined		x	integer	Determined by geometry manager
window		y	integer	Determined by geometry manager
work_area_menu				
work_in_progress_box				

[1]Gadget variant is not supported.

B.26 Scroll Bar

Table B–27 shows the controls, reasons, and arguments supported by the scroll bar widget.

Table B–27 Scroll Bar

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	create	background_color	color	Default background color
	destroy	background_pixmap	pixmap	Null
	drag	border_color	color	Default foreground color
	help	border_pixmap	pixmap	Null
	page_dec	border_width	integer	1 pixel
	page_inc	direction_r_to_l	boolean	False
	to_bottom	foreground_color	color	Default foreground color
	to_top	height	integer	height of parent - 17 units (vertical orientation); height of parent + 17 units (horizontal orientation)
	unit_dec	highlight_color	color	Default foreground color
	unit_inc	highlight_pixmap	pixmap	Null
	value_ changed	mapped_when_ managed	boolean	True
		max_value	integer	100
		min_value	integer	0
		orientation	integer	DwtOrientationVertical
		scroll_bar_inc	integer	10 units
		scroll_bar_page_inc	integer	10 units
		scroll_bar_value	integer	0
		sensitive	boolean	True
		show_arrows	boolean	True
		shown	integer	10 units
		slider_color	color	Default foreground color
		slider_pixmap	pixmap	Null

(continued on next page)

Table B–27 (Cont.) Scroll Bar

Controls	Reasons	Arguments		
		Name	**UIL Data Type**	**Default Value**
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		width	integer	width of parent + 17 units (vertical orientation); width of parent - 17 units (horizontal orientation)
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.27 Scroll Window

Table B–28 shows the controls, reasons, and arguments supported by the scroll window widget.

Table B–28 Scroll Window

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
attached_dialog_box	create	background_color	color	Default background color
caution_box	destroy	background_pixmap	pixmap	Null
color_mix	help	border_color	color	Default foreground color
command_window		border_pixmap	pixmap	Null
compound_str_txt		border_width	integer	1 pixel
dialog_box		direction_r_to_l	boolean	False
file_selection		foreground_color	color	Default foreground color
help_box		height	integer	Height of work area window + height of horizontal scroll bar
label[1]		horizontal_scroll_bar	object reference	Null
list_box		mapped_when_managed	boolean	True
main_window		sensitive	boolean	True
menu_bar		shown_value_automatic_horiz	boolean	True
message_box		shown_value_automatic_vert	boolean	True
option_menu		translations	translation_table	Default translation table syntax
popup_attached_db		user_data	any	Null
popup_dialog_box		vertical_scroll_bar	object reference	Null
popup_menu		width	integer	Width of work area window + width of vertical scroll bar
pulldown_entry[1]		work_window	object reference	Null
pulldown_menu		x	integer	Determined by geometry manager
push_button[1]		y	integer	Determined by geometry manager
radio_box				

[1]Gadget variant is not supported.

(continued on next page)

Table B–28 (Cont.) Scroll Window

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
scale				
scroll_bar				
scroll_window				
selection				
separator[1]				
simple_text				
toggle_button[1]				
user_defined				
window				
work_area_menu				
work_in_progress_box				

[1]Gadget variant is not supported.

B.28 Selection

Table B–29 shows the controls, reasons, and arguments supported by the selection widget.

Table B–29 Selection

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
attached_dialog_box	activate	accelerators	translation_table	Null
caution_box	cancel	auto_unmanage	boolean	True
color_mix	create	auto_unrealize	boolean	False
command_window	destroy	background_color	color	Default background color
compound_str_txt	focus	background_pixmap	pixmap	Null
dialog_box	help	border_color	color	Default foreground color
file_selection	map	border_pixmap	pixmap	Null
help_box	no_match	border_width	integer	1 pixel
label[1]	unmap	cancel_label	compound_string	"Cancel"
list_box		default_position	boolean	False
main_window		direction_r_to_l	boolean	False
menu_bar		file_to_extern_proc	any	Null
message_box		file_to_intern_proc	any	Null
option_menu		font_argument	font_table	Default system font
popup_attached_db		height	integer	5 pixels or as large as needed to hold children
popup_dialog_box		items	string_table	Null
popup_menu		label_label	compound_string	"Items"
pulldown_entry[1]		mapped_when_managed	boolean	True
pulldown_menu		margin_height	integer	2 pixels
push_button[1]		margin_width	integer	2 pixels
radio_box		must_match	boolean	False
scale		no_resize	boolean	True
scroll_bar		ok_label	compound_string	"Ok"
scroll_window		resize	integer	Grow only (DwtResizeGrowOnly)

[1]Gadget variant is not supported.

(continued on next page)

Table B–29 (Cont.) Selection

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
selection		resize_mode	integer	Grow only (DwtResizeGrowOnly)
separator[1]		selection_label	compound_ string	"Selection"
simple_text		selection_value	compound_ string	Null
toggle_button[1]		sensitive	boolean	True
user_defined		style	integer	Modal (DwtModal)
window		take_focus	boolean	True (modal); False (modeless)
work_area_menu		text_cols	integer	30
work_in_progress_box		title	compound_ string	"Open"
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		visible_items_count	integer	8
		width	integer	5 pixels or as large as needed to hold children
		x	integer	Centered in parent
		y	integer	Centered in parent

[1]Gadget variant is not supported.

B.29 Separator Widget

Table B–30 shows the controls, reasons, and arguments supported by the separator widget.

Table B–30 Separator Widget

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	create	background_color	color	Default background color
	destroy	background_pixmap	pixmap	Null
	help	border_color	color	Default foreground color
		border_pixmap	pixmap	Null
		border_width	integer	0 pixels
		direction_r_to_l	boolean	False
		foreground_color	color	Default foreground color
		height	integer	Height of parent (vertical orientation); 1 pixel (horizontal orientation)
		mapped_when_managed	boolean	True
		margin_bottom	integer	0 pixels
		margin_height	integer	0 pixels
		margin_left	integer	0 pixels
		margin_right	integer	0 pixels
		margin_top	integer	0 pixels
		margin_width	integer	0 pixels
		orientation	integer	DwtOrientationHorizontal
		sensitive	boolean	False
		translations	translation_table	Default translation table syntax
		user_data	any	Null
		width	integer	1 pixel (vertical orientation); width of parent (horizontal orientation)
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.30 Separator Gadget

Table B–31 shows the controls, reasons, and arguments supported by the separator gadget.

Table B–31 Separator Gadget

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	create	border_width	integer	0 pixels
	destroy	height	integer	Height of parent (vertical orientation); 1 pixel (horizontal orientation)
	help	orientation	integer	DwtOrientationHorizontal
		sensitive	boolean	False
		width	integer	1 pixel (vertical orientation); width of parent (horizontal orientation)
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.31 Simple Text

Table B–32 shows the controls, reasons, and arguments supported by the simple text widget.

Table B–32 Simple Text

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	activate	auto_show_insertion_point	boolean	True
	create	background_color	color	Default background color
	destroy	background_pixmap	pixmap	Null
	focus	blink_rate	integer	500 milliseconds
	help	border_color	color	Default foreground color
	lost_focus	border_pixmap	pixmap	Null
	value_changed	border_width	integer	1 pixel
		cols	integer	1
		editable	boolean	True
		font_argument	font_table	Default system font
		foreground_color	color	Default foreground color
		half_border	boolean	True
		height	integer	(rows * font height) + (2 * margin height)
		insertion_point_visible	boolean	True
		insertion_position	integer	0
		mapped_when_managed	boolean	True
		margin_height	integer	2 pixels
		margin_width	integer	2 pixels
		max_length	integer	MAXINT; the largest number that can fit in a C long ((2**31)-1, or 2,147,483,647)
		pending_delete	boolean	True
		resize_height	boolean	True
		resize_width	boolean	True
		rows	integer	20
		scroll_horizontal	boolean	False
		scroll_left_side	boolean	False
		scroll_top_side	boolean	False

(continued on next page)

Table B–32 (Cont.) Simple Text

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
		scroll_vertical	boolean	False
		sensitive	boolean	True
		simple_text_value	string	Null
		top_position	integer	0
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		width	integer	(columns * average character width) + (2 * margin width)
		word_wrap	boolean	False
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.32 Toggle Button Widget

Table B–33 shows the controls, reasons, and arguments supported by the toggle button widget.

Table B–33 Toggle Button Widget

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	arm	accelerator_text	compound_ string	Null
	create	alignment	integer	DwtAlignmentCenter
	destroy	background_color	color	Default background color
	disarm	background_pixmap	pixmap	Null
	help	border_color	color	Default foreground color
	value_ changed	border_pixmap	pixmap	Null
		border_width	integer	0 pixels
		button_accelerator	string	Null
		conform_to_text	boolean	True (if widget created without width and height); False (if widget created with width and height)
		direction_r_to_l	boolean	False
		font_argument	font_table	Default system font
		foreground_color	color	Default foreground color
		height[1]	integer	0 pixels
		highlight_color	color	Default foreground color
		highlight_pixmap	pixmap	Null
		indicator	boolean	True (text); False (pixmap)
		insensitive_pixmap_off	pixmap	Null
		insensitive_pixmap_on	pixmap	Null
		label_label	compound_ string	Object name
		label_label_type	integer	DwtCString
		label_pixmap	pixmap	Null
		mapped_when_ managed	boolean	True
		margin_bottom	integer	1 pixel

[1] For the widget variant, the default value for the height and width arguments depends on the setting of the conform_to_text argument. If conform_to_text is false, height and width are 0 pixels. If conform_to_text is true, height and width are as large as needed to hold the label text.

(continued on next page)

Table B–33 (Cont.) Toggle Button Widget

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
		margin_height	integer	2 pixels (text); 0 pixels (pixmap)
		margin_left	integer	9 pixels
		margin_right	integer	7 pixels
		margin_top	integer	2 pixels
		margin_width	integer	2 pixels (text); 0 pixels (pixmap)
		pixmap_off	pixmap	Null
		pixmap_on	pixmap	Null
		sensitive	boolean	True
		shape	integer	DwtRectangular
		spacing	integer	4 pixels
		toggle_value	boolean	False
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		visible_when_off	boolean	True
		widget_direction	integer	False
		width[1]	integer	0 pixels
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

[1] For the widget variant, the default value for the height and width arguments depends on the setting of the conform_to_text argument. If conform_to_text is false, height and width are 0 pixels. If conform_to_text is true, height and width are as large as needed to hold the label text.

B.33 Toggle Button Gadget

Table B–34 shows the controls, reasons, and arguments supported by the toggle button gadget.

Table B–34 Toggle Button Gadget

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	create	accelerator_text	compound_ string	Null
	destroy	alignment	integer	DwtAlignmentCenter
	help	border_width	integer	0 pixels
	value_ changed	button_accelerator	string	Null
		direction_r_to_l	boolean	False
		height	integer	0 pixels
		label_label	compound_ string	Object name
		sensitive	boolean	True
		shape	integer	DwtRectangular
		toggle_value	boolean	False
		widget_direction	integer	False
		width	integer	0 pixels
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.34 User Defined

Table B–35 shows the controls, reasons, and arguments supported by the user defined widget.

Table B–35 User Defined

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
attached_dialog_box	All reasons are supported	All arguments are supported		
caution_box				
color_mix				
command_window				
compound_str_txt				
dialog_box				
file_selection				
help_box				
label[1]				
list_box				
main_window				
menu_bar				
message_box				
option_menu				
popup_attached_db				
popup_dialog_box				
popup_menu				
pulldown_entry[1]				
pulldown_menu				
push_button[1]				
radio_box				
scale				
scroll_bar				
scroll_window				
selection				
separator[1]				
simple_text				

[1]Gadget variant is not supported.

(continued on next page)

Table B–35 (Cont.) User Defined

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
toggle_button[1]				
user_defined				
window				
work_area_menu				
work_in_progress_box				

[1]Gadget variant is not supported.

B.35 Window

Table B–36 shows the controls, reasons, and arguments supported by the window widget.

Table B–36 Window

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	create	background_color	color	Default background color
	destroy	background_pixmap	pixmap	Null
	expose	border_color	color	Default foreground color
	help	border_pixmap	pixmap	Null
		border_width	integer	1 pixel
		direction_r_to_l	boolean	False
		height	integer	0 pixels
		mapped_when_managed	boolean	True
		sensitive	boolean	True
		translations	translation_table	Default translation table syntax
		user_data	any	Null
		width	integer	0 pixels
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.36 Work Area Menu

Table B–37 shows the controls, reasons, and arguments supported by the work area menu widget.

Table B–37 Work Area Menu

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
label[1]	create	adjust_margin	boolean	False
pulldown_entry[1]	destroy	background_color	color	Default background color
push_button[1]	entry	background_pixmap	pixmap	Null
separator[1]	help	border_color	color	Default foreground color
toggle_button[1]	map	border_pixmap	pixmap	Null
user_defined	unmap	border_width	integer	1 pixel
		change_vis_atts	boolean	True
		direction_r_to_l	boolean	False
		entry_alignment	integer	True
		entry_border_width	integer	0
		font_argument	font_table	Default system font
		height	integer	16 pixels or sum of entry window heights + spacing, or 10 pixels if there are no entry windows (vertical orientation); height of widest entry window, or 5 pixels if there are no entry windows (horizontal orientation)
		mapped_when_managed	boolean	True
		margin_height	integer	3 pixels
		margin_width	integer	3 pixels
		menu_alignment	boolean	True
		menu_entry_class	class_rec_name	Null
		menu_extend_last_row	boolean	True
		menu_help_widget	object reference	True
		menu_history	object reference	Null
		menu_is_homogeneous	boolean	True (radio boxes); False (all others)

[1]Widget and gadget variants are supported.

(continued on next page)

Table B–37 (Cont.) Work Area Menu

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
		menu_num_columns	integer	1 row or column (depending on orientation)
		menu_packing	integer	DwtmenuPackingColumn (radio boxes); DwtmenuPackingTight (all others)
		menu_radio	boolean	True (radio boxes); False (all others)
		orientation	integer	DwtOrientationVertical
		radio_always_one	boolean	True
		sensitive	boolean	True
		spacing	integer	0
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		width	integer	If orientation is vertical, width is the maximum entry width or 16 pixels. If orientation is horizontal, width is the sum of width and spacing or 16 pixels.
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager

B.37 Work-in-progress Box

Table B–38 shows the controls, reasons, and arguments supported by the work-in-progress box widget.

Table B–38 Work-in-progress Box

Controls	Reasons	Arguments		
		Name	UIL Data Type	Default Value
No children are supported	cancel	accelerators	translation_table	Null
	create	auto_unmanage	boolean	True
	destroy	auto_unrealize	boolean	False
	focus	background_color	color	Default background color
	help	background_pixmap	pixmap	Null
	map	border_color	color	Default foreground color
	unmap	border_pixmap	pixmap	Null
		border_width	integer	1 pixel
		cancel_label	compound_string	"Cancel"
		default_position	boolean	False
		direction_r_to_l	boolean	False
		font_argument	font_table	Default system font
		height	integer	5 pixels or as large as needed to hold children
		icon_pixmap	pixmap	Null
		label_alignment	integer	DwtAlignmentCenter
		label_label	compound_string	Object name
		mapped_when_managed	boolean	True
		margin_height	integer	2 font units
		margin_width	integer	2 font units
		no_resize	boolean	No (no window manager resize button)
		resize	integer	DwtResizeGrowOnly
		resize_mode	integer	DwtResizeGrowOnly
		second_label	compound_string	Null
		second_label_alignment	integer	DwtAlignmentBeginning
		sensitive	boolean	True

(continued on next page)

Table B–38 (Cont.) Work-in-progress Box

Controls	Reasons	Arguments		
		Name	**UIL Data Type**	**Default Value**
		style	integer	DwtModal
		take_focus	boolean	True (modal); False (modeless)
		title	compound_ string	Object name
		translations	translation_ table	Default translation table syntax
		user_data	any	Null
		width	integer	5 pixels or as large as needed to hold children
		x	integer	Determined by geometry manager
		y	integer	Determined by geometry manager
		yes_label	compound_ string	"Yes" (DwtSYes)

B.38 UIL Arguments

Table B–39 is an alphabetical listing of all XUI Toolkit arguments (attributes) supported by UIL. For each argument, Table B–39 shows the argument name in the MIT C and VAX bindings. For your convenience, the argument's data type is repeated in this table. See the *VMS DECwindows Toolkit Routines Reference Manual* for a description of each of the supported arguments.

Table B–39 UIL Arguments

UIL Name	UIL Data Type	VAX Binding Name	MIT C Binding Name
about_label	compound_ string	DWT$C_NABOUT_LABEL	DwtNaboutLabel
accelerator_text	compound_ string	DWT$C_NACCELERATOR_TEXT	DwtNacceleratorText
accelerators	translation_ table	DWT$C_NACCELERATORS	DwtNaccelerators
accept_focus	boolean	DWT$C_NACCEPT_FOCUS	DwtNacceptFocus
adb_bottom_ attachment	integer	DWT$C_NADB_BOTTOM_ ATTACHMENT	DwtNadbBottomAttachment
adb_bottom_ offset	integer	DWT$C_NADB_BOTTOM_OFFSET	DwtNadbBottomOffset
adb_bottom_ position	integer	DWT$C_NADB_BOTTOM_POSITION	DwtNadbBottomPosition
adb_bottom_ widget	object reference	DWT$C_NADB_BOTTOM_WIDGET	DwtNadbBottomWidget
adb_left_ attachment	integer	DWT$C_NADB_LEFT_ATTACHMENT	DwtNadbLeftAttachment
adb_left_offset	integer	DWT$C_NADB_LEFT_OFFSET	DwtNadbLeftOffset
adb_left_ position	integer	DWT$C_NADB_LEFT_POSITION	DwtNadbLeftPosition
adb_left_widget	object reference	DWT$C_NADB_LEFT_WIDGET	DwtNadbLeftWidget
adb_right_ attachment	integer	DWT$C_NADB_RIGHT_ATTACHMENT	DwtNadbRightAttachment
adb_right_offset	integer	DWT$C_NADB_RIGHT_OFFSET	DwtNadbRightOffset
adb_right_ position	integer	DWT$C_NADB_RIGHT_POSITION	DwtNadbRightPosition
adb_right_ widget	object reference	DWT$C_NADB_RIGHT_WIDGET	DwtNadbRightWidget
adb_top_ attachment	integer	DWT$C_NADB_TOP_ATTACHMENT	DwtNadbTopAttachment
adb_top_offset	integer	DWT$C_NADB_TOP_OFFSET	DwtNadbTopOffset
adb_top_ position	integer	DWT$C_NADB_TOP_POSITION	DwtNadbTopPosition

(continued on next page)

Table B–39 (Cont.) UIL Arguments

UIL Name	UIL Data Type	VAX Binding Name	MIT C Binding Name
adb_top_widget	object reference	DWT$C_NADB_TOP_WIDGET	DwtNadbTopWidget
add_topic_label	compound_ string	DWT$C_NADDTOPIC_LABEL	DwtNaddtopicLabel
adjust_margin	boolean	DWT$C_NADJUST_MARGIN	DwtNadjustMargin
alignment	integer	DWT$C_NALIGNMENT	DwtNalignment
application_ name	compound_ string	DWT$C_NAPPLICATION_NAME	DwtNapplicationName
apply_label	compound_ string	DWT$C_NAPPLY_LABEL	DwtNapplyLabel
auto_show_ insertion_point	boolean	DWT$C_NAUTO_SHOW_INSERT_ POINT	DwtNautoShowInsertPoint
auto_unmanage	boolean	DWT$C_NAUTO_UNMANAGE	DwtNautoUnmanage
auto_unrealize	boolean	DWT$C_NAUTO_UNREALIZE	DwtNautoUnrealize
back_blue_ value	color	DWT$C_NBACK_BLUE_VALUE	DwtNbackBlueValue
back_green_ value	color	DWT$C_NBACK_GREEN_VALUE	DwtNbackGreenValue
back_red_value	color	DWT$C_NBACK_RED_VALUE	DwtNbackRedValue
background_ color	color	DWT$C_NBACKGROUND	DwtNbackground
background_ pixmap	pixmap	DWT$C_NBACKGROUND_PIXMAP	DwtNbackgroundPixmap
bad_frame_ message	compound_ string	DWT$C_NBADFRAME_MESSAGE	DwtNbadframeMessage
bad_library_ message	compound_ string	DWT$C_NBADLIB_MESSAGE	DwtNbadlibMessage
bidirectional_ cursor	boolean	DWT$C_NBIDIRECTIONAL_CURSOR	DwtNbidirectionalCursor
black_label	compound_ string	DWT$C_NBLACK_LABEL	DwtNblackLabel
blink_rate	integer	DWT$C_NBLINK_RATE	DwtNblinkRate
blue_label	compound_ string	DWT$C_NBLUE_LABEL	DwtNblueLabel
border_color	color	DWT$C_NBORDER_COLOR	DwtNborderColor
border_highlight	boolean	DWT$C_NBORD_HIGHLIGHT	DwtNbordHighlight
border_pixmap	pixmap	DWT$C_NBORDER_PIXMAP	DwtNborderPixmap
border_width	integer	DWT$C_NBORDER_WIDTH	DwtNborderWidth
button_ accelerator	string	DWT$C_NBUTTON_ACCELERATOR	DwtNbuttonAccelerator

(continued on next page)

Table B–39 (Cont.) UIL Arguments

UIL Name	UIL Data Type	VAX Binding Name	MIT C Binding Name
cache_help_ library	boolean	DWT$C_NCACHE_HELP_LIBRARY	DwtNcacheHelpLibrary
cancel_button	object reference	DWT$C_NCANCEL_BUTTON	DwtNcancelButton
cancel_label	compound_ string	DWT$C_NCANCEL_LABEL	DwtNcancelLabel
change_vis_atts	boolean	DWT$C_NCHANGE_VIS_ATTS	DwtNchangeVisAtts
child_overlap	boolean	DWT$C_NCHILD_OVERLAP	DwtNchildOverlap
close_label	compound_ string	DWT$C_NCLOSE_LABEL	DwtNcloseLabel
color_model	integer	DWT$C_NCOLOR_MODEL	DwtNcolorModel
cols	integer	DWT$C_NCOLS	DwtNcols
command_ value	string	DWT$C_NVALUE	DwtNvalue
compound_ text_value	compound_ string	DWT$C_NVALUE	DwtNvalue
conform_to_text	boolean	DWT$C_NCONFORM_TO_TEXT	DwtNconformToText
copy_label	compound_ string	DWT$C_NCOPY_LABEL	DwtNcopyLabel
create_ horizontal_ scroll_bar	boolean	DWT$C_NHORIZONTAL	DwtNhorizontal
decimal_points	integer	DWT$C_NDECIMAL_POINTS	DwtNdecimalPoints
default_button	object reference	DWT$C_NDEFAULT_BUTTON	DwtNdefaultButton
default_ horizontal_ offset	integer	DWT$C_NDEFAULT_HORIZONTAL_ OFFSET	DwtNdefaultHorizontalOffset
default_position	boolean	DWT$C_NDEFAULT_POSITION	DwtNdefaultPosition
default_ pushbutton	integer	DWT$C_NDEFAULT_PUSHBUTTON	DwtNdefaultPushbutton
default_vertical_ offset	integer	DWT$C_NDEFAULT_VERTICAL_ OFFSET	DwtNdefaultVerticalOffset
dialog_font	font_table	DWT$C_NFONT	DwtNfont
dir_mask	compound_ string	DWT$C_NDIR_MASK	DwtNdirMask
direction_r_to_l	boolean	DWT$C_NDIRECTION_R_TO_L	DwtNdirectionRToL
dismiss_label	compound_ string	DWT$C_NDISMISS_LABEL	DwtNdismissLabel
disp_win_ margin	integer	DWT$C_NDISP_WIN_MARGIN	DwtNdispWinMargin

(continued on next page)

Table B–39 (Cont.) UIL Arguments

UIL Name	UIL Data Type	VAX Binding Name	MIT C Binding Name
display_col_ win_height	integer	DWT$C_NDISPLAY_COL_WIN_HEIGHT	DwtNdisplayColWinHeight
display_col_ win_width	integer	DWT$C_NDISPLAY_COL_WIN_WIDTH	DwtNdisplayColWinWidth
display_label	compound_ string	DWT$C_NDISPLAY_LABEL	DwtNdisplayLabel
display_window	object reference	DWT$C_NDISPLAY_WINDOW	DwtNdisplayWindow
edit_label	compound_ string	DWT$C_NEDIT_LABEL	DwtNeditLabel
editable	boolean	DWT$C_NEDITABLE	DwtNeditable
entry_alignment	integer	DWT$C_NENTRY_ALIGNMENT	DwtNentryAlignment
entry_border_ width	integer	DWT$C_NENTRY_BORDER	DwtNentryBorder
error_open_ message	compound_ string	DWT$C_NERROROPEN_MESSAGE	DwtNerroropenMessage
exit_label	compound_ string	DWT$C_NEXIT_LABEL	DwtNexitLabel
file_label	compound_ string	DWT$C_NFILE_LABEL	DwtNfileLabel
file_search_ proc	any	DWT$C_NFILE_SEARCH_PROC	DwtNfileSearchProc
file_selection_ value	compound_ string	DWT$C_NVALUE	DwtNvalue
file_to_extern_ proc	any	DWT$C_NFILE_TO_EXTERN_PROC	DwtNfileToExternProc
file_to_intern_ proc	any	DWT$C_NFILE_TO_INTERN_PROC	DwtNfileToInternProc
fill_highlight	boolean	DWT$C_NFILL_HIGHLIGHT	DwtNfillHighlight
filter_label	compound_ string	DWT$C_NFILTER_LABEL	DwtNfilterLabel
first_topic	compound_ string	DWT$C_NFIRST_TOPIC	DwtNfirstTopic
font_argument	font_table	DWT$C_NFONT	DwtNfont
foreground_ color	color	DWT$C_NFOREGROUND	DwtNforeground
fraction_base	integer	DWT$C_NFRACTION_BASE	DwtNfractionBase
full_label	compound_ string	DWT$C_NFULL_LABEL	DwtNfullLabel
glossary_label	compound_ string	DWT$C_NGLOSSARY_LABEL	DwtNglossaryLabel

(continued on next page)

Table B–39 (Cont.) UIL Arguments

UIL Name	UIL Data Type	VAX Binding Name	MIT C Binding Name
glossary_topic	compound_ string	DWT$C_NGLOSSARY_TOPIC	DwtNglossaryTopic
go_back_label	compound_ string	DWT$C_NGOBACK_LABEL	DwtNgobackLabel
go_over_label	compound_ string	DWT$C_NGOOVER_LABEL	DwtNgooverLabel
go_to_label	compound_ string	DWT$C_NGOTO_LABEL	DwtNgotoLabel
gobacktopic_ label	compound_ string	DWT$C_NGOBACKTOPIC_LABEL	DwtNgobacktopicLabel
gototopic_label	compound_ string	DWT$C_NGOTOTOPIC_LABEL	DwtNgototopicLabel
grab_merge_ translations	translation_ table	DWT$C_NGRAB_MERGE_ TRANSLATIONS	DwtNgrabMergeTranslations
gray_label	compound_ string	DWT$C_NGRAY_LABEL	DwtNgrayLabel
green_label	compound_ string	DWT$C_NGREEN_LABEL	DwtNgreenLabel
half_border	boolean	DWT$C_NHALF_BORDER	DwtNhalfBorder
height	integer	DWT$C_NHEIGHT	DwtNheight
help_ acknowledge_ label	compound_ string	DWT$C_NHELP_ACKNOWLEDGE_ LABEL	DwtNhelpAcknowledgeLabel
help_font	font_table	DWT$C_NHELP_FONT	DwtNhelpFont
help_label	compound_ string	DWT$C_NHELP_LABEL	DwtNhelpLabel
help_message_ title	compound_ string	DWT$C_NHELPMESSAGE_TITLE	DwtNhelpmessageTitle
help_message_ title_type	integer	DWT$C_NHELPMESSAGE_TITLE_ TYPE	DwtNhelpmessageTitleType
help_on_help_ title	compound_ string	DWT$C_NHELP_ON_HELP_TITLE	DwtNhelpOnHelpTitle
helphelp_label	compound_ string	DWT$C_NHELPHELP_LABEL	DwtNhelphelpLabel
helpontitle_label	compound_ string	DWT$C_NHELPONTITLE_LABEL	DwtNhelpontitleLabel
helptitle_label	compound_ string	DWT$C_NHELPTITLE_LABEL	DwtNhelptitleLabel
highlight_color	color	DWT$C_NHIGHLIGHT	DwtNhighlight
highlight_ pixmap	pixmap	DWT$C_NHIGHLIGHT_PIXMAP	DwtNhighlightPixmap

(continued on next page)

Table B–39 (Cont.) UIL Arguments

UIL Name	UIL Data Type	VAX Binding Name	MIT C Binding Name
history	string	DWT$C_NHISTORY	DwtNhistory
history_box_label	compound_string	DWT$C_NHISTORYBOX_LABEL	DwtNhistoryboxLabel
history_label	compound_string	DWT$C_NHISTORY_LABEL	DwtNhistoryLabel
hls_label	compound_string	DWT$C_NHLS_LABEL	DwtNhlsLabel
horizontal_scroll_bar	object reference	DWT$C_NHORIZONTAL_SCROLL_BAR	DwtNhorizontalScrollBar
hot_spot_pixmap	pixmap	DWT$C_NHOT_SPOT_PIXMAP	DwtNhotSpotPixmap
hue_label	compound_string	DWT$C_NHUE_LABEL	DwtNhueLabel
icon_pixmap	pixmap	DWT$C_NICON_PIXMAP	DwtNiconPixmap
indicator	boolean	DWT$C_NINDICATOR	DwtNindicator
insensitive_pixmap	pixmap	DWT$C_NINSENSITIVE_PIXMAP	DwtNinsensitivePixmap
insensitive_pixmap_off	pixmap	DWT$C_NINSENSITIVE_PIXMAP_OFF	DwtNinsensitivePixmapOff
insensitive_pixmap_on	pixmap	DWT$C_NINSENSITIVE_PIXMAP_ON	DwtNinsensitivePixmapOn
insertion_point_visible	boolean	DWT$C_NINSERTION_POINT_VISIBLE	DwtNinsertionPointVisible
insertion_position	integer	DWT$C_NINSERTION_POSITION	DwtNinsertionPosition
items	string_table	DWT$C_NITEMS	DwtNitems
keyword_label	compound_string	DWT$C_NKEYWORD_LABEL	DwtNkeywordLabel
keywords_label	compound_string	DWT$C_NKEYWORDS_LABEL	DwtNkeywordsLabel
label_alignment	integer	DWT$C_NLABEL_ALIGNMENT	DwtNlabelAlignment
label_label	compound_string	DWT$C_NLABEL	DwtNlabel
label_label_type	integer	DWT$C_NLABEL_TYPE	DwtNlabelType
label_pixmap	pixmap	DWT$C_NPIXMAP	DwtNpixmap
library_spec	compound_string	DWT$C_NLIBRARY_SPEC	DwtNlibrarySpec
library_type	integer	DWT$C_NLIBRARY_TYPE	DwtNlibraryType
light_label	compound_string	DWT$C_NLIGHT_LABEL	DwtNlightLabel

(continued on next page)

Table B–39 (Cont.) UIL Arguments

UIL Name	UIL Data Type	VAX Binding Name	MIT C Binding Name
lines	integer	DWT$C_NLINES	DwtNlines
list_updated	boolean	DWT$C_NLIST_UPDATED	DwtNlistUpdated
main_ command_ window	object reference	DWT$C_NCOMMAND_WINDOW	DwtNcommandWindow
main_ horizontal_ scroll_bar	object reference	DWT$C_NHORIZONTAL_SCROLL_BAR	DwtNhorizontalScrollBar
main_label	compound_ string	DWT$C_NMAIN_LABEL	DwtNmainLabel
main_menu_bar	object reference	DWT$C_NMENU_BAR	DwtNmenuBar
main_vertical_ scroll_bar	object reference	DWT$C_NVERTICAL_SCROLL_BAR	DwtNverticalScrollBar
main_work_ window	object reference	DWT$C_NWORK_WINDOW	DwtNworkWindow
mapped_when_ managed	boolean	DWT$C_NMAPPED_WHEN_MANAGED	DwtNmappedWhenManaged
margin_bottom	integer	DWT$C_NMARGIN_BOTTOM	DwtNmarginBottom
margin_height	integer	DWT$C_NMARGIN_HEIGHT	DwtNmarginHeight
margin_left	integer	DWT$C_NMARGIN_LEFT	DwtNmarginLeft
margin_right	integer	DWT$C_NMARGIN_RIGHT	DwtNmarginRight
margin_top	integer	DWT$C_NMARGIN_TOP	DwtNmarginTop
margin_width	integer	DWT$C_NMARGIN_WIDTH	DwtNmarginWidth
mask_to_ extern_proc	any	DWT$C_NMASK_TO_EXTERN_PROC	DwtNmaskToExternProc
mask_to_ intern_proc	any	DWT$C_NMASK_TO_INTERN_PROC	DwtNmaskToInternProc
match_colors	boolean	DWT$C_NMATCH_COLORS	DwtNmatchColors
max_length	integer	DWT$C_NMAX_LENGTH	DwtNmaxLength
max_value	integer	DWT$C_NMAX_VALUE	DwtNmaxValue
menu_ alignment	boolean	DWT$C_NMENU_ALIGNMENT	DwtNmenuAlignment
menu_entry_ class	class_rec_name	DWT$C_NMENU_ENTRY_CLASS	DwtNmenuEntryClass
menu_extend_ last_row	boolean	DWT$C_NMENU_EXTEND_LAST_ROW	DwtNmenuExtendLastRow
menu_help_ widget	object reference	DWT$C_NMENU_HELP_WIDGET	DwtNmenuHelpWidget
menu_history	object reference	DWT$C_NMENU_HISTORY	DwtNmenuHistory

(continued on next page)

Table B–39 (Cont.) UIL Arguments

UIL Name	UIL Data Type	VAX Binding Name	MIT C Binding Name
menu_is_ homogeneous	boolean	DWT$C_NMENU_IS_HOMOGENEOUS	DwtNmenuIsHomogeneous
menu_num_ columns	integer	DWT$C_NMENU_NUM_COLUMNS	DwtNmenuNumColumns
menu_packing	integer	DWT$C_NMENU_PACKING	DwtNmenuPacking
menu_radio	boolean	DWT$C_NMENU_RADIO	DwtNmenuRadio
menu_type	integer	DWT$C_NMENU_TYPE	DwtNmenuType
min_value	integer	DWT$C_NMIN_VALUE	DwtNminValue
mixer_label	compound_ string	DWT$C_NMIXER_LABEL	DwtNmixerLabel
mixer_window	object reference	DWT$C_NMIXER_WINDOW	DwtNmixerWindow
must_match	boolean	DWT$C_NMUST_MATCH	DwtNmustMatch
new_blue_value	color	DWT$C_NNEW_BLUE_VALUE	DwtNnewBlueValue
new_disp_ window	object reference	DWT$C_NNEW_DISP_WINDOW	DwtNnewDispWindow
new_green_ value	color	DWT$C_NNEW_GREEN_VALUE	DwtNnewGreenValue
new_red_value	color	DWT$C_NNEW_RED_VALUE	DwtNnewRedValue
no_keyword_ message	compound_ string	DWT$C_NNOKEYWORD_MESSAGE	DwtNnokeywordMessage
no_label	compound_ string	DWT$C_NNO_LABEL	DwtNnoLabel
no_resize	boolean	DWT$C_NNO_RESIZE	DwtNnoResize
no_title_ message	compound_ string	DWT$C_NNOTITLE_MESSAGE	DwtNnotitleMessage
null_library_ message	compound_ string	DWT$C_NNULLLIB_MESSAGE	DwtNnulllibMessage
null_topic_ message	compound_ string	DWT$C_NNULLTOPIC_MESSAGE	DwtNnulltopicMessage
ok_label	compound_ string	DWT$C_NOK_LABEL	DwtNokLabel
option_label	compound_ string	DWT$C_NOPTION_LABEL	DwtNoptionLabel
orientation	integer	DWT$C_NORIENTATION	DwtNorientation
orig_blue_value	color	DWT$C_NORIG_BLUE_VALUE	DwtNorigBlueValue
orig_disp_ window	object reference	DWT$C_NORIG_DISP_WINDOW	DwtNorigDispWindow
orig_green_ value	color	DWT$C_NORIG_GREEN_VALUE	DwtNorigGreenValue
orig_red_value	color	DWT$C_NORIG_RED_VALUE	DwtNorigRedValue

(continued on next page)

Table B–39 (Cont.) UIL Arguments

UIL Name	UIL Data Type	VAX Binding Name	MIT C Binding Name
overview_topic	compound_ string	DWT$C_NOVERVIEW_TOPIC	DwtNoverviewTopic
pending_delete	boolean	DWT$C_NPENDING_DELETE	DwtNpendingDelete
pixmap_off	pixmap	DWT$C_NPIXMAP_OFF	DwtNpixmapOff
pixmap_on	pixmap	DWT$C_NPIXMAP_ON	DwtNpixmapOn
prompt	compound_ string	DWT$C_NPROMPT	DwtNprompt
radio_always_ one	boolean	DWT$C_NRADIO_ALWAYS_ONE	DwtNradioAlwaysOne
red_label	compound_ string	DWT$C_NRED_LABEL	DwtNredLabel
reset_label	compound_ string	DWT$C_NRESET_LABEL	DwtNresetLabel
resizable	boolean	DWT$C_NRESIZABLE	DwtNresizable
resize	integer	DWT$C_NRESIZE	DwtNresize
resize_height	boolean	DWT$C_NRESIZE_HEIGHT	DwtNresizeHeight
resize_mode	integer	DWT$C_NRESIZE	DwtNresize
resize_width	boolean	DWT$C_NRESIZE_WIDTH	DwtNresizeWidth
rgb_label	compound_ string	DWT$C_NRGB_LABEL	DwtNrgbLabel
rows	integer	DWT$C_NROWS	DwtNrows
rubber_ positioning	boolean	DWT$C_NRUBBER_POSITIONING	DwtNrubberPositioning
sat_label	compound_ string	DWT$C_NSAT_LABEL	DwtNsatLabel
save_as_label	compound_ string	DWT$C_NSAVEAS_LABEL	DwtNsaveasLabel
scale_height	integer	DWT$C_NSCALE_HEIGHT	DwtNscaleHeight
scale_value	integer	DWT$C_NVALUE	DwtNvalue
scale_width	integer	DWT$C_NSCALE_WIDTH	DwtNscaleWidth
scroll_bar_inc	integer	DWT$C_NINC	DwtNinc
scroll_bar_ page_inc	integer	DWT$C_NPAGE_INC	DwtNpageInc
scroll_bar_value	integer	DWT$C_NVALUE	DwtNvalue
scroll_horizontal	boolean	DWT$C_NSCROLL_HORIZONTAL	DwtNscrollHorizontal
scroll_left_side	boolean	DWT$C_NSCROLL_LEFT_SIDE	DwtNscrollLeftSide
scroll_top_side	boolean	DWT$C_NSCROLL_TOP_SIDE	DwtNscrollTopSide
scroll_vertical	boolean	DWT$C_NSCROLL_VERTICAL	DwtNscrollVertical

(continued on next page)

Table B–39 (Cont.) UIL Arguments

UIL Name	UIL Data Type	VAX Binding Name	MIT C Binding Name
search_apply_label	compound_string	DWT$C_NSEARCHAPPLY_LABEL	DwtNsearchapplyLabel
search_keyword_box_label	compound_string	DWT$C_NSEARCHKEYWORDBOX_LABEL	DwtNsearchkeywordboxLabel
search_label	compound_string	DWT$C_NSEARCH_LABEL	DwtNsearchLabel
search_title_box_label	compound_string	DWT$C_NSEARCHTITLEBOX_LABEL	DwtNsearchtitleboxLabel
second_label	compound_string	DWT$C_NSECOND_LABEL	DwtNsecondLabel
second_label_alignment	integer	DWT$C_NSECOND_LABEL_ALIGNMENT	DwtNsecondLabelAlignment
select_all_label	compound_string	DWT$C_NSELECTALL_LABEL	DwtNselectallLabel
selected_items	string_table	DWT$C_NSELECTED_ITEMS	DwtNselectedItems
selection_label	compound_string	DWT$C_NSELECTION_LABEL	DwtNselectionLabel
selection_value	compound_string	DWT$C_NVALUE	DwtNvalue
sensitive	boolean	DWT$C_NSENSITIVE	DwtNsensitive
set_new_color_proc	any	DWT$C_NSET_NEW_COLOR_PROC	DwtNsetNewColorProc
shadow	boolean	DWT$C_NSHADOW	DwtNshadow
shape	integer	DWT$C_NSHAPE	DwtNshape
show_arrows	boolean	DWT$C_NSHOW_ARROWS	DwtNshowArrows
show_value	boolean	DWT$C_NSHOW_VALUE	DwtNshowValue
show_value_automatic	boolean	DWT$C_NSHOW_VALUE_AUTOMATIC	DwtNshowValueAutomatic
shown	integer	DWT$C_NSHOWN	DwtNshown
shown_value_automatic_horiz	boolean	DWT$C_NSHOWN_VALUE_AUTOMATIC_HORIZ	DwtNshownValueAutomaticHoriz
shown_value_automatic_vert	boolean	DWT$C_NSHOWN_VALUE_AUTOMATIC_VERT	DwtNshownValueAutomaticVert
simple_text_value	string	DWT$C_NVALUE	DwtNvalue
single_selection	boolean	DWT$C_NSINGLE_SELECTION	DwtNsingleSelection
slider_color	color	DWT$C_NSLIDER	DwtNslider
slider_label	compound_string	DWT$C_NSLIDER_LABEL	DwtNsliderLabel

(continued on next page)

Table B–39 (Cont.) UIL Arguments

UIL Name	UIL Data Type	VAX Binding Name	MIT C Binding Name
slider_pixmap	pixmap	DWT$C_NSLIDER_PIXMAP	DwtNsliderPixmap
spacing	integer	DWT$C_NSPACING	DwtNspacing
style	integer	DWT$C_NSTYLE	DwtNstyle
take_focus	boolean	DWT$C_NTAKE_FOCUS	DwtNtakeFocus
text_cols	integer	DWT$C_NTEXT_COLS	DwtNtextCols
text_merge_ translations	translation_ table	DWT$C_NTEXT_MERGE_ TRANSLATIONS	DwtNtextMergeTranslations
text_translation	translation_ table	DWT$C_NT_TRANSLATION	DwtNtTranslation
title	compound_ string	DWT$C_NTITLE	DwtNtitle
title_label	compound_ string	DWT$C_NTITLE_LABEL	DwtNtitleLabel
titles_label	compound_ string	DWT$C_NTITLES_LABEL	DwtNtitlesLabel
toggle_value	boolean	DWT$C_NVALUE	DwtNvalue
top_position	integer	DWT$C_NTOP_POSITION	DwtNtopPosition
topic_titles_ label	compound_ string	DWT$C_NTOPICTITLES_LABEL	DwtNtopictitlesLabel
translations	translation_ table	DWT$C_NTRANSLATIONS	DwtNtranslations
units	integer	DWT$C_NUNITS	DwtNunits
user_data	any	DWT$C_NUSER_DATA	DwtNuserData
value_label	compound_ string	DWT$C_NVALUE_LABEL	DwtNvalueLabel
vertical_scroll_ bar	object reference	DWT$C_NVERTICAL_SCROLL_BAR	DwtNverticalScrollBar
view_label	compound_ string	DWT$C_NVIEW_LABEL	DwtNviewLabel
visible_items_ count	integer	DWT$C_NVISIBLE_ITEMS_COUNT	DwtNvisibleItemsCount
visible_when_ off	boolean	DWT$C_NVISIBLE_WHEN_OFF	DwtNvisibleWhenOff
visit_glossary_ label	compound_ string	DWT$C_NVISITGLOS_LABEL	DwtNvisitglosLabel
visit_label	compound_ string	DWT$C_NVISIT_LABEL	DwtNvisitLabel
visittopic_label	compound_ string	DWT$C_NVISITTOPIC_LABEL	DwtNvisittopicLabel

(continued on next page)

Table B–39 (Cont.) UIL Arguments

UIL Name	UIL Data Type	VAX Binding Name	MIT C Binding Name
white_label	compound_ string	DWT$C_NWHITE_LABEL	DwtNwhiteLabel
widget_direction	integer	DWT$C_NDIRECTION_R_TO_L	DwtNdirectionRToL
width	integer	DWT$C_NWIDTH	DwtNwidth
word_wrap	boolean	DWT$C_NWORD_WRAP	DwtNwordWrap
work_window	object reference	DWT$C_NWORK_WINDOW	DwtNworkWindow
x	integer	DWT$C_NX	DwtNx
y	integer	DWT$C_NY	DwtNy
yes_label	compound_ string	DWT$C_NYES_LABEL	DwtNyesLabel

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	————	Local Digital subsidiary or approved distributor
Internal[1]	————	USASSB Order Processing - WMO/E15 *or* U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

[1]For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

Reader's Comments

VMS DECwindows User Interface Language Reference Manual
AA–MG22B–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

-- Do Not Tear - Fold Here and Tape ----------

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01-3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here ----------

Reader's Comments

VMS DECwindows User Interface Language Reference Manual
AA–MG22B–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

_____

What I like best about this manual is _____

_____

_____

What I like least about this manual is _____

_____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

- Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35 110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

— Do Not Tear - Fold Here