Application Developer’s Guide
### Table of Contents

**Preface** vii  
What’s in this Manual vii  
Conventions Used in this Manual ix  
Technical Support x

**Chapter 1: Accessing the Operating System** 1  
Manipulating Environment Variables 2  
Manipulating OpenVMS Logicals and Symbols 5  
Accessing the Operating System Using SPAWN 7  
Changing the Current Working Directory 13

**Chapter 2: Interapplication Communication for UNIX and OpenVMS** 15  
Methods of Interapplication Communication 15  
Choosing the Best Method 17  
Interapplication Communication Using SPAWN 19  
Executing PV-WAVE Commands Externally 22  
Using LINKNLOAD to Call External Programs 27  
Calling PV-WAVE in a Statically Linked Program 37  
Using wavevars() to Access PV-WAVE Variables 50  
Special Considerations for Noninteractive Applications 57  
Communication with Remote Procedure Calls (UNIX Only) 59  
Remote Procedure Call Examples 72  
Interapplication Communication Using the Socket OPI 86

**Chapter 3: Interapplication Communication for Windows**
93

Methods of Interapplication Communication 93
Choosing the Best Method 94
Using LINKNLOAD to Call External Programs 96
Calling PV-WAVE as a Dynamically Linked Program 98
Accessing Data in PV-WAVE Variables 109
Special Considerations for Noninteractive Applications 114
Using Dynamic Data Exchange (DDE) 116
Console Versus Windows Subsystem Applications 119

Chapter 4: Building VDA Tools 121

What Are VDA Tools? 121
A Technical Perspective on VDA Tools 124
What is the VDA Tools Manager? 126
VDA Tool Ingredients 130
VDA Utilities 139
Connecting to Online Help 140
Tool-to-Tool Communication Routines 141
Creating Optional Areas in VDA Tools 142
Creating a Navigator 146
Resources and Strings 148
Using Resources and Strings in PV-WAVE Applications 150
Adding Online Help 152

Chapter 5: Using WAVE Widgets 157

Methods of GUI Programming in PV-WAVE 157
Introduction to WAVE Widgets 161
First Example and Basic Steps 162
Initializing WAVE Widgets 164
Adding Event Handlers 223
Adding Timers 224
Adding Work Procedures 226
Adding Input Handler Procedures (Motif Only) 226
Changing the Cursor 227
Creating Tables 228
Running an Application 228
Related Include Files 229
Example Widget Toolbox Application 229
Programming Tips and Cautions 231

Chapter 7: WAVE Widgets Reference 233

Chapter 8: Widget Toolbox Reference 363

Chapter 9: VDA Tools Manager API (Tm) 413

Chapter 10: Graphical Elements API (Tm) 473

Chapter 11: VDA Tools Utilities (Wo) 499

Chapter 12: Localizing PV-WAVE Applications 581
  Localizing the VDA Tools 581
  Localizing the PV-WAVE Home Window 584
  Localizing the PV-WAVE Print Dialog Box 586

Appendix A: Motif Widget Classes A-1
  Motif Widget Classes A-1
  Convenience Widgets A-3
Appendix B: Motif Callback Parameters  B-1
Motif Callback Parameters  B-1

Appendix C: Widget Toolbox Cursors  C-1
Standard X Cursors  C-1
Custom Cursors  C-4

Appendix D: Developing Portable Applications  D-1
Writing Portable WAVE Widgets Applications  D-1
Writing Portable VDA Tools  D-6
Writing System-specific Code  D-8
Example Code  D-9

Appendix E: Virtual Keys  E-1

Appendix F: Windows Color and Font Support  F-1
Predefined Colors for Windows Systems  F-1
Setting Fonts on Windows  F-3

Index 1
Preface

This manual is for application developers who want to create Visual Data Analysis applications with a Graphical User Interface (GUI). A wide range of tools and technologies are available to the PV-WAVE® application developer. Choosing the best tools for your application is your first challenge. This manual contains introductory information on interapplication communication, WAVE Widgets, the Widget Toolbox, and VDA Tools. The detailed information you need to use these technologies is in this manual as well.

What’s in this Manual

This manual covers the following topics:

• **Chapter 1, Accessing the Operating System** — Discusses the ways in which you can manipulate environment variables, logicals, and symbols from within PV-WAVE. In addition, the SPAWN command is introduced as a way to execute external programs from within PV-WAVE. Finally, ways to change the current directory are discussed.

• **Chapter 2, Interapplication Communication for UNIX and OpenVMS** — Discusses a variety of methods for interapplication communication. For example, PV-WAVE can execute external programs and exchange data with them. In addition, external programs can call PV-WAVE to perform graphics, data manipulation, and other functions.

• **Chapter 3, Interapplication Communication for Windows** — Discusses a variety of methods for interapplication communication available under Windows.
• **Chapter 4, Building VDA Tools** — Describes VDA Tool architecture and outlines a method for building new VDA Tools.

• **Chapter 5, Using WAVE Widgets** — Discusses how to create a Motif GUI using the WAVE Widgets functions.

• **Chapter 6, Using the Widget Toolbox** — Discusses how to create a Motif GUI using the Widget Toolbox functions.

• **Chapter 7, WAVE Widgets Reference** — Detailed description of the WAVE Widgets functions.

• **Chapter 8, Widget Toolbox Reference** — Detailed description of the Widget Toolbox Functions.

• **Chapter 9, VDA Tools Manager API (Tm)** — Detailed description of the basic set of Tools Manager API functions.

• **Chapter 10, Graphical Elements API (Tm)** — Detailed description of the Tools Manager functions used specifically to create and manipulate Graphical Elements.

• **Chapter 11, VDA Tools Utilities (Wo)** — Detailed description of the VDA Tools Utilities, a set of convenience routines used to develop a VDA Tool user interface quickly and efficiently.

• **Chapter 12, Localizing PV-WAVE Applications** — Explains how to localize VDA Tools, the Home Window, and related applications.

• **Appendix A, Motif Widget Classes** — Lists the widget classes available under Motif.

• **Appendix B, Motif Callback Parameters** — Lists the required callback parameters for widget routines under Motif.

• **Appendix C, Widget Toolbox Cursors** — Lists the standard and custom cursors that are available for use with the WtCursor function under Motif and Windows.

• **Appendix D, Developing Portable Applications** — Discusses several portability issues, particularly with respect to WAVE Widgets applications.

• **Appendix E, Virtual Keys** — Lists the symbolic constant names, hexadecimal values, and keyboard equivalents for the virtual-key codes used by the Microsoft Windows operating system.

• **Appendix F, Windows Color and Font Support** — Discusses color and font support for WAVE Widgets on Windows platforms.

• **Index** — A subject index with hypertext links to information in this manual.
Conventions Used in this Manual

You will find the following conventions used throughout this manual:

- Code examples appear in this typeface. For example:

  PLOT, temp, s02, Title = ‘Air Quality’

- Code comments are shown in this typeface, immediately below the commands they describe. For example:

  PLOT, temp, s02, Title = ‘Air Quality’
  ; This command plots air temperature data vs. sulphur dioxide concentration.

- Variables are shown in lowercase italics (myvar), function and procedure names are shown in uppercase (XYOUTS), keywords are shown in mixed case italic (XTitle), and system variables are shown in regular mixed case type (!Version). For better readability, all application development routines are shown in mixed case (WwMainMenu).

- A $ at the end of a line of PV-WAVE code indicates that the current statement is continued on the following line. By convention, use of the continuation character ($) in this document reflects its syntactically correct use in PV-WAVE. This means, for instance, that strings are never split onto two lines without the addition of the string concatenation operator (+). For example, the following lines would produce an error if entered literally in PV-WAVE.

  WAVE> PLOT, x, y, Title = 'Average $
  Air Temperatures by Two-Hour Periods'
  ; Note that the string is split onto two lines; an error
  ; message is displayed if you enter a string this way.

  The correct way to enter these lines is:

  WAVE> PLOT, x, y, Title = 'Average ' + $
  'Air Temperatures by Two-Hour Periods'
  ; This is the correct way to split a string onto two command lines.

- Reserved words, such as FOR, IF, CASE, are always shown in uppercase.
Technical Support

If you have problems installing, unlocking, or running your software, contact Visual Numerics Technical Support by calling:

<table>
<thead>
<tr>
<th>Office Location</th>
<th>Phone Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corporate Headquarters</td>
<td>713-784-3131</td>
</tr>
<tr>
<td>Houston, Texas</td>
<td>303-939-8920</td>
</tr>
<tr>
<td>Boulder, Colorado</td>
<td>+33-1-46-93-94-20</td>
</tr>
<tr>
<td>France</td>
<td>+49-711-13287-0</td>
</tr>
<tr>
<td>Germany</td>
<td>+81-3-5211-7760</td>
</tr>
<tr>
<td>Japan</td>
<td>+82-2-3273-2633</td>
</tr>
<tr>
<td>Korea</td>
<td>+52-5-514-9730</td>
</tr>
<tr>
<td>Mexico</td>
<td>+886-2-727-2255</td>
</tr>
<tr>
<td>Taiwan</td>
<td>+44-1-344-458-700</td>
</tr>
</tbody>
</table>

Users outside the U.S., France, Germany, Japan, Korea, Mexico, Taiwan, and the U.K. can contact their local agents.

Please be prepared to provide the following information when you call for consultation during Visual Numerics business hours:

- Your license number, a six-digit number that can be found on the packing slip accompanying this order. (If you are evaluating the software, just mention that you are from an evaluation site.)
- The name and version number of the product. For example, PV-WAVE 7.0.
- The type of system on which the software is being run. For example, SPARCstation, IBM RS/6000, HP 9000 Series 700.
- The operating system and version number. For example, HP-UX 10.2 or IRIX 6.5.
- A detailed description of the problem.
FAX and E-mail Inquiries

Contact Visual Numerics Technical Support staff by sending a FAX to:

<table>
<thead>
<tr>
<th>Office Location</th>
<th>FAX Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corporate Headquarters</td>
<td>713-781-9260</td>
</tr>
<tr>
<td>Boulder, Colorado</td>
<td>303-245-5301</td>
</tr>
<tr>
<td>France</td>
<td>+33-1-46-93-94-39</td>
</tr>
<tr>
<td>Germany</td>
<td>+49-711-13287-99</td>
</tr>
<tr>
<td>Japan</td>
<td>+81-3-5211-7769</td>
</tr>
<tr>
<td>Korea</td>
<td>+82-2-3273-2634</td>
</tr>
<tr>
<td>Mexico</td>
<td>+52-5-514-4873</td>
</tr>
<tr>
<td>Taiwan</td>
<td>+886-2-727-6798</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>+44-1-344-458-748</td>
</tr>
</tbody>
</table>

or by sending E-mail to:

<table>
<thead>
<tr>
<th>Office Location</th>
<th>E-mail Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boulder, Colorado</td>
<td><a href="mailto:support@boulder.vni.com">support@boulder.vni.com</a></td>
</tr>
<tr>
<td>France</td>
<td><a href="mailto:support@vni-paris.fr">support@vni-paris.fr</a></td>
</tr>
<tr>
<td>Germany</td>
<td><a href="mailto:support@visual-numerics.de">support@visual-numerics.de</a></td>
</tr>
<tr>
<td>Japan</td>
<td><a href="mailto:vda-sprt@vnij.co.jp">vda-sprt@vnij.co.jp</a></td>
</tr>
<tr>
<td>Korea</td>
<td><a href="mailto:support@vni.co.kr">support@vni.co.kr</a></td>
</tr>
<tr>
<td>Taiwan</td>
<td><a href="mailto:support@vni.com.tw">support@vni.com.tw</a></td>
</tr>
<tr>
<td>United Kingdom</td>
<td><a href="mailto:support@vniuk.co.uk">support@vniuk.co.uk</a></td>
</tr>
</tbody>
</table>
# Electronic Services

<table>
<thead>
<tr>
<th>Service</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>General e-mail</td>
<td><a href="mailto:info@boulder.vni.com">info@boulder.vni.com</a></td>
</tr>
<tr>
<td>Support e-mail</td>
<td><a href="mailto:support@boulder.vni.com">support@boulder.vni.com</a></td>
</tr>
<tr>
<td>World Wide Web</td>
<td><a href="http://www.vni.com">http://www.vni.com</a></td>
</tr>
<tr>
<td>Anonymous FTP</td>
<td>ftp.boulder.vni.com</td>
</tr>
<tr>
<td>PV-WAVE Mailing List:</td>
<td><a href="mailto:Majordomo@boulder.vni.com">Majordomo@boulder.vni.com</a></td>
</tr>
<tr>
<td>To subscribe</td>
<td>subscribe pv-wave YourEmailAddress</td>
</tr>
<tr>
<td>To post messages</td>
<td><a href="mailto:pv-wave@boulder.vni.com">pv-wave@boulder.vni.com</a></td>
</tr>
</tbody>
</table>
Accessing the Operating System

This chapter discusses the methods for communicating with the operating system from PV-WAVE. The main topics include:

- Manipulating environment variables (UNIX/Windows)
- Manipulating OpenVMS logicals and symbols
- Using SPAWN to access the operating system
- Changing the current directory

The utility routines used to communicate with specific operating systems are summarized in the following table:

<table>
<thead>
<tr>
<th>Method</th>
<th>UNIX</th>
<th>OpenVMS</th>
<th>Win</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPAWN</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Lets you spawn a child process to execute commands. The output generated by the commands can be captured in a variable for later processing by PV-WAVE.</td>
</tr>
<tr>
<td>GETENV, SETENV, ENVIRONMENT</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Manipulate environment variables.</td>
</tr>
</tbody>
</table>
NOTE It is important to realize that environment variables are not a PV-WAVE feature, they are part of every UNIX and Windows process. Although they can serve as a form of global memory, it is best to avoid using them in that way. Instead, system variables and common blocks should be used in that role. This will make your code portable to OpenVMS systems.

UNIX Environment Variables

Every UNIX process has an “environment”. The environment consists of “environment variables”, each of which has a string value associated with it. Some environment variables always exist, such as \texttt{path}, which tells the shell where to look for programs, or \texttt{term}, which specifies the kind of terminal being used. You may add other environment variables at any time from an interactive shell, or you can add them to the file, such as the \texttt{.login} file, that is executed when you log in.

When a process is created, it is given a copy of the environment from its parent process. PV-WAVE is no exception to this; when started, it inherits a copy of its parent’s environment. The parent process to PV-WAVE is usually the interactive shell from which it was started. In turn, any child process created by PV-WAVE (such as those from the SPAWN procedure) inherits a copy of PV-WAVE’s current environment.
Environment variables should be used for communicating with child processes. For example, you can change the SHELL environment variable prior to calling SPAWN. SPAWN then uses the newly defined shell to run its process.

PV-WAVE provides the following procedures and functions for manipulating the UNIX environment. For more information on these procedures and functions, see the PV-WAVE Reference.

**Windows Environment Variables**

Environment variables define particular characteristics of your operating system environment, such as important directory paths. For example, when PV-WAVE is installed, several environment variables are automatically defined, such as the %VNI_DIR% variable, which points to the main directory where Visual Numerics products are installed.

Whenever a process is created, it is given a copy of the environment from its parent process. PV-WAVE is no exception to this; when started, it inherits a copy of its parent’s environment. The parent process to PV-WAVE is usually the Command window from which it was started. In turn, any child process created by PV-WAVE (such as those from the SPAWN procedure) inherits a copy of PV-WAVE’s current environment.

PV-WAVE provides the following procedures and functions for manipulating environment variables. For more information on these procedures and functions, see the PV-WAVE Reference.

**SETENV: Adding a New Environment Variable**

The SETENV procedure adds a new environment variable, or changes the value of an existing environment variable, in the PV-WAVE process. It has the form:

```
SETENV, environment_expression
```

where `environment_expression` is a scalar string containing an environment expression to be added to the environment.

**UNIX Example**

For example, you can change the shell used by SPAWN by changing the value of the SHELL environment variable. A statement to change to the Bourne shell /bin/sh is:

```
SETENV, ‘SHELL=/bin/sh’
```
**Windows Example**

For example, to change the HOMEPATH environment variable to point to the directory `D:\users\chris\utah_data`:

```
SETENV, 'HOMEPATH=D:\users\chris\utah_data'
```

**GETENV: Getting an Environment Variable’s Equivalence String**

The GETENV function returns the value (equivalence string) of a specified environment variable. It has the form:

```
result = GETENV(name)
```

where `name` is the name of the environment variable for which the value is desired. If `name` does not exist in the environment, a null string is returned.

**UNIX Example**

For example, to determine the type of terminal being used, you can enter the statement:

```
PRINT, 'The terminal type is:', GETENV('TERM')
```

Executing this statement on a Sun workstation gives the result:

```
The terminal type is: sun
```

**Windows Example**

For example, to determine value of the HOMEPATH variable, you can enter the statement:

```
PRINT, 'The value of HOMEPATH is:', $GETENV('HOMEPATH')
```

Executing this statement gives the result:

```
The value of HOMEPATH is: D:\users\chris
```

**ENVIRONMENT: Getting the Values of All Environment Variables**

The ENVIRONMENT function returns a string array containing the values (equivalence strings) of all the environment variables currently found in the PV-WAVE process environment.
**UNIX Example**

The following statements print the entire environment, one environment variable per line:

```
WAVE> env = ENVIRONMENT() ; Get a copy of the environment.
WAVE> FOR I = 0, N_ELEMENTS(env)-1 $ DO PRINT, env(I)  ; Print out, one variable per line.
```

**Windows Example**

The following statements print the entire environment, one environment variable per line:

```
WAVE> env = ENVIRONMENT() ; Get a copy of the environment.
WAVE> FOR I = 0, N_ELEMENTS(env)-1 $ DO PRINT, env(I)  ; Print out, one variable per line.
```

---

**Manipulating OpenVMS Logicals and Symbols**

PV-WAVE provides the following procedures and functions for manipulating OpenVMS logicals and symbols. For more information on these procedures and functions, see the PV-WAVE Reference.

**SETLOG: Defining a New Logical**

The SETLOG procedure defines a logical name. It has the form:

```
SETLOG, lognam, value
```

where *lognam* is the scalar string containing the name of the logical to be defined and *value* is a string giving the value to which the logical will be set. If *value* is a string array, *lognam* is defined as a multi-valued logical where each element of *value* defines one of the equivalence strings.

**TRNLOG: Getting a Logical’s Equivalence String**

The TRNLOG function searches the OpenVMS name tables for a specified logical name and returns the equivalence string(s) in a variable. TRNLOG returns the
OpenVMS status code associated with the translation as a longword value. As with all OpenVMS status codes, success is indicated by an odd value (least significant bit is set) and failure by an even value. It has the form:

\[
result = \text{TRNLOG}(\text{lognam}, \text{value})
\]

where \text{lognam} is a scalar string containing the name of the logical to be translated and \text{value} is a named variable into which the equivalence string is placed. If \text{lognam} has more than one equivalence string, the first one is used.

The following statements allow you to see these values:

WAVE> \text{ret} = \text{TRNLOG}("\text{SYSSYSROOT}", \text{trans}, \$
\text{/Full, /Issue_Error})
\text{;} \text{Translate the logical.}
WAVE> \text{print, trans}

**DELLOG: Deleting a Logical**

The DELLOG procedure deletes a logical name. It has the form:

\[
\text{DELLOG, lognam}
\]

where \text{lognam} is a scalar string containing the name of the logical to be deleted.

**SET_SYMBOL: Defining a Symbol**

The SET_SYMBOL procedure defines a DCL (Digital Command Language) interpreter symbol for the current process. It has the form:

\[
\text{SET_SYMBOL, name, value}
\]

where \text{name} is a scalar string containing the name of the symbol to be defined and \text{value} is a scalar string containing the value with which \text{name} will be defined.

**GET_SYMBOL: Getting a Symbol’s Value**

The GET_SYMBOL function returns the value of an OpenVMS DCL (Digital Command Language) interpreter symbol as a scalar string. If the symbol is undefined, the null string is returned. It has the form:

\[
result = \text{GET_SYMBOL(name)}
\]

where \text{name} is a string containing the name of the symbol to translate.
DELETE_SYMBOL: Deleting a Symbol

The DELETE_SYMBOL procedure deletes a DCL (Digital Command Language) interpreter symbol from the current process. It has the form:

```
DELETE_SYMBOL, name
```

where `name` is a string containing the name of the symbol to delete.

---

**Accessing the Operating System Using SPAWN**

SPAWN is a very flexible command. It can create an interactive command interpreter (shell, for UNIX users) process, or simply issue a single command and return. Under UNIX, in this second case, the command can either be passed to a shell for processing, or it can be executed directly as a child process of PV-WAVE.

Under Windows, SPAWN can create a new Command window, or simply issue a single command and return.

This section discusses SPAWN as it is used to issue commands and capture output. For detailed information on how to use SPAWN to execute and transfer data to and from a child process (external program), see Interapplication Communication Using SPAWN on page 19.

**Using SPAWN to Issue Commands**

The SPAWN procedure spawns a child process to execute a given command. It has the form:

```
SPAWN [, command [, result]]
```

The `command` parameter is a string containing the command to be issued. `command` can be either a scalar or an array.

**UNIX USERS** The way in which it is treated depends on whether the Noshell keyword is specified. For more information on Noshell, see Avoiding the Shell under UNIX on page 11.

If the `result` parameter is not present, the output from the child process simply goes to the standard output (usually the terminal). Otherwise, the output from the child process is placed into a string array (one line of output per array element) and assigned to `result`. 
Interactive Use of SPAWN

If SPAWN is called without arguments, an interactive shell (UNIX) or command interpreter (OpenVMS) or Command window (Windows) is started. You can enter one or more shell commands. While you use the shell or command interpreter process, PV-WAVE is suspended. When you exit the child process, control returns to PV-WAVE, which resumes at the point where it left off. The PV-WAVE session remains exactly as you left it.

UNIX Example

The following statements demonstrate the interactive use of SPAWN.

An example of using SPAWN in a UNIX environment is:

WAVE> SPAWN
%date
Fri Aug 26 13:55:00 MDT 1988
%exit
WAVE>

OpenVMS Example

An example of using SPAWN in an OpenVMS environment is:

WAVE> SPAWN
$ SHOW TIME
29-JAN-1990 16:32:23
$ LOGOUT
WAVE>

Windows Example

The following statements demonstrate the interactive use of SPAWN.

WAVE> SPAWN
 ; At this point, a Command window appears, and you can enter a
 ; command, such as the date command.
D:\vni\wave> date
The current date is: Fri 07/16/1993
Enter the new date: <mm-dd-yy>
D:\vni\wave> exit
 ; The exit command exits the Command window and returns you to
 ; the PV-WAVE session.
WAVE>
NOTE Using SPAWN in this manner is equivalent to using the $ command. The difference between these two is that $ can only be used interactively, while SPAWN can be used interactively or in PV-WAVE programs.

UNIX Shells

The most common UNIX shells are the Bourne shell (/bin/sh), the C shell (/bin/csh), and the Korn shell (/bin/ksh). Rather than force you to use a given shell, PV-WAVE follows the UNIX convention of using the shell specified by the UNIX environment variable SHELL. If SHELL does not exist, the Bourne shell is used. The UNIX environment is discussed in Manipulating Environment Variables on page 2 in this chapter.

Under UNIX, the interactive form of SPAWN is provided primarily for users of the Bourne shell and for compatibility with OpenVMS. Shells that offer process suspension (e.g., /bin/csh) offer a more convenient and efficient way to get the same effect.

VMS Command Interpreter

Under OpenVMS, the command interpreter is always DCL (Digital Command Language).

Avoiding the Command Prompt Window (Windows)

As mentioned above, SPAWN usually creates a command prompt window and passes the command to it, instead of simply creating a child process to directly execute the command. This default action is taken because the command prompt window provides useful actions such as wildcard expansion and argument processing. Although this is usually desirable, creating a command prompt window process has the drawback of being slower than necessary; it simply takes longer to start a command prompt window. However, it is possible to avoid using the command prompt window by using the Noshell keyword.

When SPAWN is called and Noshell is present and non-zero, the command is executed as a direct child process, avoiding the extra overhead of starting a command prompt window. This is faster, but since there is no command prompt window to break the command into separate arguments, you have to do it. Every Windows NT program is called with a series of arguments. When you issue a command prompt window command, you separate the arguments with white space (blanks and tabs). The command prompt window then breaks up the command into an array of argu-
ments, and calls the command (the first word of the command), passing it the array of arguments.

Thus, if you use *Noshell* to avoid using a command prompt window, you have to break up the arguments yourself. In this case, the *command* argument should be a string array. The first element of the array is the name of the command to use, and the following elements contain the arguments.

For example, consider the command:

```
WAVE> SPAWN, 'dir c:\reports /a:h'
```

To issue this command without a shell, you write it as:

```
WAVE> SPAWN, ['dir' 'c:\reports' '/a:h'], /Noshell
```

**Non-interactive Use of SPAWN**

If SPAWN is called with a single argument, that argument is taken as a command to be executed. In this case, PV-WAVE starts a child process and passes the command to it. The argument should be a scalar string. The child process executes the command and exits, at which point PV-WAVE resumes operation. This form of operation is very convenient for executing single commands from PV-WAVE programs. For example, it is sometimes useful to delete a temporary scratch file. SPAWN can be used as shown in the following program fragment.

**UNIX/OpenVMS Example**

```
OPENW, Unit, 'scratch.dat', /Get_Lun
    ; Open the scratch file. Use the Get_Lun keyword to allocate a file unit.

    ; PV-WAVE commands that use the file go here.
FREE_LUN, Unit
    ; Deallocate the file unit and close the file.
if (!Version.os EQ 'vms') THEN $
    Cmd = 'DELETE' else Cmd = 'rm'
    ; Use the !Version system variable to determine the proper file deletion command for the current operating system.
SPAWN, Cmd + 'scratch.dat'
    ; Delete the file using SPAWN.
```

**Windows Example**

```
OPENW, Unit, 'scratch.dat', /Get_Lun
```
; Open the scratch file. Use the Get_Lun keyword to allocate a file
; unit.

. . .

; PV-WAVE commands that use the file go here.

FREE_LUN, Unit
; Deallocate the file unit and close the file.

SPAWN, 'ERASE scratch.dat'
; Delete the file using SPAWN.

NOTE  Actually, the Delete keyword to the OPEN procedure is a more efficient
way to handle this job. The above examples serve only to demonstrate the use of
the SPAWN procedure.

Avoiding the Shell under UNIX

As mentioned above, SPAWN usually creates a shell process and passes the
command to this shell, instead of simply creating a child process to directly execute
the command. This default action is taken because the shell provides useful actions
such as wildcard expansion and argument processing. Although this is usually
desirable, creating a shell process has the drawback of being slower than necessary;
it simply takes longer to start a shell. However, it is possible to avoid using the shell
by using the Noshell keyword.

When SPAWN is called and Noshell is present and non-zero, the command is exe-
cuted as a direct child process, avoiding the extra overhead of starting a shell. This
is faster, but since there is no shell to break the command into separate arguments,
you have to do it. Every UNIX program is called with a series of arguments. When
you issue a shell command, you separate the arguments with white space (blanks
and tabs). The shell then breaks up the command into an array of arguments, and
calls the command (the first word of the command), passing it the array of
arguments.

Thus, if you use Noshell to avoid using a shell, you have to break up the arguments
yourself. In this case, the command argument should be a string array. The first ele-
ment of the array is the name of the command to use, and the following elements
contain the arguments.

For example, consider the command:

WAVE> SPAWN, 'ps ax'

which uses the UNIX ps command to show running processes on the computer. To
issue this command without a shell, you write it as:
Capturing Output

By default, any output generated by the spawned command is sent to the standard output, which is usually the terminal. It is possible to capture this output in a string array by calling SPAWN with a second argument. If this second argument, called `result`, is present, all output from the child process is put into a string array, one line of output per array element, and is assigned to `result`.

**UNIX/OpenVMS Example**

For example, the following statements can be used to give a simplistic count of the number of users logged onto the computer:

```plaintext
if (!Version.os EQ 'vms') THEN $
    Cmd = 'SHOW USERS' ELSE Cmd = 'who'
    ; Use the !Version system variable to determine the command
    ; to use.

SPAWN, Cmd, Users
    ; Issue the command; catch the result in a string array.

N = N_ELEMENTS(Users)
    ; Count how many lines of output cue back. Under UNIX, this is the
    ; number of users logged in.

if (!Version.os EQ 'vms') THEN N = N - 5
    ; OpenVMS outputs five extra header lines which are not actual
    ; users.

PRINT, 'There are ', N, ' users logged on.'
    ; Give the result.
```

**Windows Example**

For example, the following statements can be used to capture the current environment variable settings in a variable:

```plaintext
SPAWN, 'set', vars
    ; Issue the command; catch the result in a string array.

PRINT, vars
    ; List the environment variables in the PV-WAVE Console window.
```
Changing the Current Working Directory

Like every process, PV-WAVE has a current working directory. This is the default directory that is used whenever you specify a file without explicitly supplying the directory. The initial working directory is the directory you were in when you issued the command to start PV-WAVE.

Using the CD Procedure

You can use the CD command to change this working directory at any point during the PV-WAVE session. This new working directory affects the current session, and any child processes that you start from PV-WAVE, but it does not change the current directory of the process that started PV-WAVE. Therefore, when you exit PV-WAVE, you will find yourself back in the directory you were in when you started.

On a UNIX system, to change the current directory to /usr/stardata, enter:
WAVE> CD, '/usr/stardata'

On an OpenVMS system, to change the current directory to SYS$SYSDEVICE:[STARDATA], enter:
WAVE> CD, 'SYS$SYSDEVICE:[STARDATA]'

On a Windows system, to change the current directory to \home\stardata, enter:
WAVE> CD, '\home\stardata'

In order to change to your login directory, you can provide a null argument. In addition, the Current keyword can be used to save the current directory before any change is made. The following command saves the current directory and then changes it to your home directory:
WAVE> CD, '', Current=OLDDIR

Later, you can restore the current directory to its previous value with the command:
WAVE> CD, OLDDIR

Using the PUSHD, POPD, and PRINTD Procedures

The PUSHD, POPD, and PRINTD procedures are provided to make interactive use of CD more convenient by maintaining a stack of directories. PUSHD saves the current directory on the top of the stack and then changes it to the directory given by its argument. POPD sets the current directory to the directory at the top of the
stack and removes that directory from the stack. PRINTD shows you the current entries on the stack.

**UNIX/OpenVMS Example**

Using these user procedures, the previous CD example could be written:

```
WAVE> PUSHD, ''
        ; Change to your home directory.

. . .
        ; Execute some statements.
WAVE> POPD
        ; Return to the original working directory.
```

**Windows Example**

Using these user procedures, the previous CD example could be written:

```
WAVE> PUSHD, 'c:\temp'
        ; Change the directory.

. . .
        ; Execute some statements.
WAVE> POPD
        ; Return to the original working directory.
```
Interapplication Communication for UNIX and OpenVMS

PV-WAVE provides a variety of methods for interapplication communication under UNIX and OpenVMS. For example:

- PV-WAVE can execute external programs and exchange data with them. Depending on the method used, the exchange of data can be unidirectional (one-way) or bidirectional (two-way).
- External programs can call PV-WAVE to perform graphics, data manipulation, and other functions. Again, depending on the method used, the communication can be unidirectional or bidirectional.

Methods of Interapplication Communication

The following table summarizes the methods of interapplication communication that can be used between PV-WAVE and other external applications. This table and the following section, Choosing the Best Method on page 17, can help you to determine the most appropriate method of interapplication communication to accomplish a desired task. Each method listed is described in detail later in this chapter.
## Routines for Accessing the Operating System

<table>
<thead>
<tr>
<th>Method</th>
<th>UNIX</th>
<th>OpenVMS</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPAWN</td>
<td>Yes</td>
<td>Yes</td>
<td>A system routine that executes an external program from within PV-WAVE. Allows data to be transferred to and from PV-WAVE via bidirectional pipes and PV-WAVE’s standard I/O facilities. See page 19.</td>
</tr>
<tr>
<td>waveinit, wavecmd, waveterm</td>
<td>Yes</td>
<td>No</td>
<td>Routines that allow a C or FORTRAN program to start PV-WAVE, execute commands, and exit PV-WAVE. No data is transferred back to the calling program. Available on UNIX systems only. See page 22.</td>
</tr>
<tr>
<td>LINKNLOAD</td>
<td>Yes *</td>
<td>Yes</td>
<td>A system routine that allows PV-WAVE to call an external function via dynamic linked libraries. It is the simplest method for calling your own C code from PV-WAVE. Allows the transfer of binary data. Data is transferred between the C program and PV-WAVE via the wavevars routine. See page 27.</td>
</tr>
<tr>
<td>cwavec</td>
<td>Yes</td>
<td>Yes</td>
<td>A routine that allows a statically linked C program to access PV-WAVE. Data is transferred between the C program and PV-WAVE via the wavevars routine. See page 37.</td>
</tr>
<tr>
<td>cwavefor</td>
<td>Yes</td>
<td>Yes</td>
<td>Works like cwavec, except from a statically linked FORTRAN program. See page 37.</td>
</tr>
<tr>
<td>Option Programming Interface (OPI)</td>
<td>Yes</td>
<td>Yes</td>
<td>For developers who want to create optional modules that can be loaded explicitly by any PV-WAVE user. These optional modules can be written in C or FORTRAN, and can contain new system functions or other primitives. See the PV-WAVE Programmer’s Guide for more information.</td>
</tr>
<tr>
<td>CALL_UNIX</td>
<td>Yes</td>
<td>No</td>
<td>A system routine that uses Remote Procedure Call (RPC) technology to allow PV-WAVE to call a separate application across a UNIX network. See page 72.</td>
</tr>
<tr>
<td>CALL_WAVE</td>
<td>Yes</td>
<td>No</td>
<td>A system routine that uses Remote Procedure Call (RPC) technology to allow a separate application to call PV-WAVE across a UNIX network. See page 72.</td>
</tr>
</tbody>
</table>
Choosing the Best Method

It is important to select the most appropriate method of interapplication communication for your particular needs. Choosing the wrong method often requires much more work than is necessary to accomplish a given task.

This section describes typical scenarios where some kind of interapplication communication is required. After each scenario is described (in italics), a suitable solution for interapplication communication is suggested.

I’m running PV-WAVE, and I want to execute an external program I’ve written. I’m not really concerned about returning anything to PV-WAVE.

This is the simplest case of interapplication communication. The SPAWN procedure is the best choice. SPAWN executes an external program, or an operating system command, from PV-WAVE. (Although it is not a requirement in this scenario, SPAWN can return data from the external application to PV-WAVE.)

For information on using SPAWN, see Interapplication Communication Using SPAWN on page 19.

I’m on a UNIX system, and I just want to call PV-WAVE from my C or FORTRAN program, execute some PV-WAVE commands, and exit PV-WAVE. It isn’t necessary for my program to retrieve any data from PV-WAVE.

The routines waveinit, wavecmd, and waveterm can be used to accomplish this sort of task. These routines, which are only available under UNIX, allow one-way (unidirectional) communication from a C or FORTRAN application to PV-WAVE. They start PV-WAVE, execute specified commands, and exit PV-WAVE. No data is transferred back to the calling program.

For information on using waveinit, wavecmd, and waveterm, see Executing PV-WAVE Commands Externally on page 22.

Routines for Accessing the Operating System (Continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>UNIX</th>
<th>OpenVMS</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket OPI</td>
<td>Yes</td>
<td>Yes</td>
<td>Allows you to treat network connections as streams of bytes that can be read from or written to. With the Socket OPI, you can write client and server applications entirely in PV-WAVE. See page 86.</td>
</tr>
</tbody>
</table>

* Not currently available for all versions of UNIX.
I wrote a C program, and I want to be able to link it dynamically with PV-WAVE. My program needs to be able to access data directly from the data space of PV-WAVE. When my program is finished running, I want control returned back to PV-WAVE.

The LINKNLOAD procedure is the simplest method for attaching your own code to PV-WAVE. LINKNLOAD is a system procedure that calls a function in an external sharable object. When used in conjunction with the `wavevars` function, data can be passed back and forth between the user-written routine and PV-WAVE.

For information on LINKNLOAD, see Using LINKNLOAD to Call External Programs on page 27. For information on the data transfer function `wavevars` see Using `wavevars()` to Access PV-WAVE Variables on page 50.

**NOTE** See also the section Method 2: The Option Programming Interface on page 29. The Option Programming Interface (OPI) functions allow user-written code to access PV-WAVE variables and use other PV-WAVE functionality. OPI provides greater flexibility and control than `wavevars`.

I want to be able to call PV-WAVE from a C or FORTRAN program I’ve written. I want the program to be statically linked with PV-WAVE.

The `cwavec` function allows a statically linked C program to access PV-WAVE’s data space. Data is transferred between the C program and PV-WAVE via the `wavevars` routine. In addition, the `cwavefor` function allows a statically linked FORTRAN program to access PV-WAVE’s data space.

For information on `cwavec` and `cwavefor` see Calling PV-WAVE in a Statically Linked Program on page 37. For information on the data transfer function `wavevars` see Using `wavevars()` to Access PV-WAVE Variables on page 50.

I have an application running across the UNIX network that I want my PV-WAVE program to communicate with.

Under UNIX, Remote Procedure Calls (RPCs) can be used to facilitate this kind of communication.

For information on interapplication communication routines that support RPCs, see Remote Procedure Call Examples on page 72.
Interapplication Communication Using SPAWN

This section explains how to use SPAWN to communicate with a child process (external program).

Communicating with a Child Process

In the previous chapter, the SPAWN procedure was used to start a child process and execute PV-WAVE commands. The PV-WAVE process waited until the child process was finished before continuing. The communication was one-way and only a single “transaction” was completed.

It is also possible to start a child process using SPAWN and continue the PV-WAVE process without waiting for the child process to finish. To do this, PV-WAVE attaches a bidirectional pipe to the standard input and output of the child process. This pipe appears in the PV-WAVE process as a normal logical file unit. Once a process has been started in this way, the normal PV-WAVE Input/Output facilities are used to communicate with it. The ability to use a child process in this manner allows you to solve specialized problems using other languages, and to take advantage of existing programs.

Starting the Child Process

In order to start such a process, the Unit keyword is used with SPAWN to specify a named variable into which the logical file unit number will be stored. Once the child process has done its work, the FREE_LUN procedure is used to close the pipe and delete the process.

When using a child process in this manner, it is important to understand the following points:

- The EOF function always returns false when applied to a pipe. This means that it is not possible to use this function to know when the child process is finished. As a result, the child process must be written in such a way that the controlling PV-WAVE procedure knows how much data to send and how much is coming back.

- A UNIX pipe is simply a buffer maintained by the operating system. It has a fixed length, and can therefore become completely filled. When this happens, the operating system puts the process that is filling the pipe to sleep until the process at the other end consumes the buffered data. The use of a bidirectional pipe can therefore lead to deadlock situations in which both processes are wait-
ing for each other. This can happen if the parent and child processes do not synchronize their reading and writing activities.

- Most C programs use the input/output facilities provided by the Standard C Library stdio. In situations where PV-WAVE and the child process are carrying on a running dialog (as opposed to a single transaction), the normal buffering performed by stdio on the output file can cause communications to hang. We recommend calling the stdio function setbuf() as the first statement of the child program to eliminate such buffering:

\[(\text{void}) \text{setbuf}(\text{stdout}, (\text{char} *) 0);\]

It is important that this statement occur before any output operation is executed, otherwise it will have no effect.

**Example: Communicating with a Child Process Using SPAWN**

This example assumes you have a C program, test_pipe.c, that accepts floating point values from its standard input and returns their average on the standard output. The code for such a C program is shown next. An explanation of the program is given immediately after the listing.

You can find the following example file in:

$WAVE_DIR/demo/interapp/spawn/test_pipe.c

```c
#include <stdio.h>
/* System error number */
extern int errno;
/* System error messages */
extern char *sys_errlist[];
/* Length of sys_errlist*/
extern int sys_nerr;
main()
{
    float *data, total = 0.0;
    long i, n;
    /* Make sure the output is not buffered */
    setbuf(stdout, (char *) 0);
    /* Find out how many points */
    if (!fread(&n, sizeof(long), 1, stdin)) goto error;
    /* Get memory for the array */
    if (!(data = (float *) malloc((unsigned)
                               (n * sizeof(float))))) goto error;
```
/* Read the data */
if (!fread(data, sizeof(float), n, stdin)) goto error;
/* Calculate the average */
for (i=0; i < n; i++) total += data[i];
total /= (float) n;
/* Return the answer */
if (!fwrite(&total, sizeof(float), 1, stdout)) goto error;
return;
error:
fprintf(stderr, "test_pipe: %s\n",
        sys_errlist[errno]);
}

This C program returns a single-precision floating-point value, which is the average of the input values. The program also reads a long integer that tells how many data points to expect.

Since the amount of input and output for this program is explicitly known, and because it reads all of its input at the beginning and writes all of its results at the end, a deadlock situation as described in the previous section cannot occur.

NOTE
In actual practice, such a trivial program would never be used from PV-WAVE; it is simpler and more efficient to perform the calculation within PV-WAVE. It does, however, serve to illustrate the method by which significant programs can be called from PV-WAVE.

Using SPAWN to Access the C Program from PV-WAVE

The following PV-WAVE statements use test_pipe to determine the average of the values 0 through 9:

SPAWN, 'test_pipe', Unit=unit, /Noshell
    ; Start test_pipe. The use of the Noshell keyword is not necessary,
    ; but speeds up the startup process.
WRITEU, unit, 10L, FINDGEN(10)
    ; Send the number of points followed by the actual data.
answer = 0.0
READU, unit, answer
PRINT, "Average = ", answer
FREE_LUN, unit
    ; Close the pipe, delete the child process, and deallocate the LUN.
Executing these statements gives the result:

```
Average = 4.50000
```

This mechanism provides a simple way to augment PV-WAVE with code written in other languages such as C or FORTRAN. In this case, however, it is not as efficient as writing the required operation entirely in PV-WAVE. The actual cost depends primarily on the amount of data being transferred. For example, the above example can be performed entirely in PV-WAVE using a simple statement like:

```
PRINT, ‘Average = ‘, TOTAL(FINDGEN(10))/10.0
```

The PV-WAVE calculation is always faster; however, the difference may only be significant when a large amount of data is transferred.

---

**Executing PV-WAVE Commands Externally**

**NOTE** The commands described in this section are only available for PV-WAVE running on UNIX systems.

**NOTE** The commands described in this section cannot be used with PV-WAVE in runtime mode.

The routines `waveinit`, `wavecmd`, and `waveterm` let you execute PV-WAVE from an external program, such as a C or FORTRAN program. The first routine, `waveinit`, starts PV-WAVE. The second routine, `wavecmd`, sends commands to PV-WAVE. The third routine, `waveterm`, ends the PV-WAVE session.

**NOTE** These routines allow one-way (unidirectional) communication only. That is, PV-WAVE cannot pass data back to the calling program. If you require data to be passed back to the calling program (bidirectional transfer), then choose another method of interapplication communication.

---

**Compiling the External Program**

After these routines have been incorporated into an external program, the program must be compiled with the correct object module. The object module is named `callwave.o`, and its full pathname is the following, where `arch` is the machine architecture, such as `hps700` or `sun4`:

```
$WAVE_DIR/bin/bin.arch/callwave.o
```
If you are unsure about the architecture, you can get the information by typing the following command:

\$WAVE\_DIR/bin/arch

Once you know the architecture, you can compile your program. For example, if you were on a Sun-4, you would type one of the following commands:

cc -o myprog myprog.c \$WAVE\_DIR/bin/bin.sun4/callwave.o

or

f77 -o myprog myprog.f \$WAVE\_DIR/bin/bin.sun4/callwave.o

**NOTE** Compile and link options will vary by platform, and are sometimes site specific. Refer to the man page for your compiler for more detailed information on how to compile programs on your system.

---

**Starting PV-WAVE from an External Program with waveinit**

The initialization routine \texttt{waveinit} starts PV-WAVE. The routine first checks the environment variable \texttt{WAVE\_DIR}. If \texttt{WAVE\_DIR} is defined, the routine uses the path \texttt{\$WAVE\_DIR/bin/wave} to start PV-WAVE. When \texttt{WAVE\_DIR} is not defined, the routine uses the path /usr/local/bin/wave. The last part of the path (wave) may be set to a symbolic link. When this path (/usr/local/bin/wave) is used, the path /usr/local/lib/wave must also be a valid path. The last part of this path may be set to a symbolic link that points to the main PV-WAVE directory.

The \texttt{waveinit} function has one output parameter, the name of a file to contain the PV-WAVE alphanumeric output (not the graphics output). For example, you can specify a character string denoting the filename or you can specify a null string, denoting that no alphanumeric output should be produced. Suppose you have a C program and you do not want to save the output, you can use:

\texttt{waveinit("");}

To write the alphanumeric output to a file (e.g., wave.out), use the following:

\texttt{waveinit("wave.out");}

To do the same thing from a FORTRAN program, you would use the following two commands:

\texttt{CALL WAVEINIT("");}

or

\texttt{CALL WAVEINIT(‘wave.out’);}
Sending Commands to PV-WAVE with \texttt{wavecmd}

To send commands to PV-WAVE, you must use the \texttt{wavecmd} routine. The routine’s single parameter is the string you want to send to PV-WAVE. For example, to plot a vector $[1,2,3,4,5]$ from a C program you would use:

\begin{verbatim}
wavecmd("plot, [1,2,3,4,5]");
\end{verbatim}

From a FORTRAN program:

\begin{verbatim}
CALL WAVECMD('PLOT, [1,2,3,4,5]')
\end{verbatim}

The \texttt{wavecmd} routine can be called as many times as required.

Ending the Session with PV-WAVE with \texttt{waveterm}

When you are finished sending commands to PV-WAVE, you must call the routine \texttt{waveterm}. This routine ends the session with PV-WAVE and closes the necessary files. In a C program, type:

\begin{verbatim}
waveterm();
\end{verbatim}

and in a FORTRAN program type:

\begin{verbatim}
CALL WAVETERM();
\end{verbatim}

Example: Calling PV-WAVE from a C Program

The following C code sample shows how to pass a 5-element array to PV-WAVE, have PV-WAVE perform some calculations, and produce a surface plot.

You can find the following listed file in:

$WAVE\_DIR/demo/interapp/wavecmd/example.c

\begin{verbatim}
#include <stdio.h>
#include <string.h>
define MAXSIZE 128

main()
{
/*
 * Variables for array passing
 */
    char buf[MAXSIZE];
    char temp[MAXSIZE];
/*
 * Variables for calculations
 */
\end{verbatim}
int i,x[5];

/*
 * Perform some calculations
 */
for (i=0; i<5; i++) x[i] = i * 4;

/*
 * Start PV-WAVE sending the alphanumeric
 * output to 'wave.save'.
 */
wavinit("wave.save");

/*
 * Send the commands to PV-WAVE. First, we
 * need to send the array to PV-WAVE.
 */
sprintf(buf,"a=[%d", x[0]);
for (i=1; i<5; i++) {
    sprintf(temp,",%d", x[i]);
    strcat(buf, temp);
}
strcat(buf, "]");
wavecmd(buf);

/*
 * Next, we perform a matrix multiplication.
 * Then we print the newly formed
 * two-dimensional array, as well as display
 * it as a surface.
 */
wavecmd("b = a # a");
wavecmd("print, b");
wavecmd("surface, b");

/*
 * The following WAIT is needed so that the
 * surface will not be deleted as soon as it
 * has been drawn.
 */
wavecmd("wait, 3.0");

/*
 * Since we are done sending commands to
 * PV-WAVE, we must call waveterm.
 */
waveterm();
Example: Calling PV-WAVE from a FORTRAN Program

The following FORTRAN code fragment shows how you can use PV-WAVE to manipulate and plot data.

You can find the following listed file in:
$WAVE_DIR/demo/interapp/wavecmd/examplefor.f

PROGRAM TEST
C
C Start PV-WAVE and send the alphanumeric
C output to 'wavefor.save'.
C
      CALL WAVEINIT('wavefor.save')
C
C Send my commands to PV-WAVE. First, we
C define a 5-element array. Next, we perform
C a matrix multiplication. Then we print the
C newly formed two-dimensional array and
C display it as a surface.
C
      CALL WAVECMD('A =[110,90,27,48,60]')
      CALL WAVECMD('B =A # A')
      CALL WAVECMD('PRINT, B')
      CALL WAVECMD('SURFACE, B')
C The following WAIT is needed so that the
C surface will not be deleted as soon as it
C has been drawn.
C
      CALL WAVECMD('WAIT, 3.0')
C
C Since we are done sending commands to
C PV-WAVE, we must call waveterm.
C
      CALL WAVETERM()
END
Using LINKNLOAD to Call External Programs

The LINKNLOAD function provides simplified access to external routines in shareable images. LINKNLOAD calls a function in an external sharable object and returns a scalar value. Parameters are passed through PV-WAVE to the specified external function by reference, thus allowing the external function to alter values of PV-WAVE variables. It is the simplest method for attaching your own C code to PV-WAVE. Unlike SPAWN, LINKNLOAD allows the sharing of binary data without duplication (transferal) overhead.

Usage

\[
\text{result} = \text{LINKNLOAD}(\text{object}, \text{symbol} [\text{, param}_1, ..., \text{param}_n])
\]

Parameters

object — A string specifying the filename, including the file path (use ./<filename> if the file is in the current directory), of the sharable object file to be linked and loaded.

symbol — A string specifying the function name (symbol entry point) to be invoked in the shared object file.

param_i — The data to be passed as a parameter to the function.

For more detailed information on the LINKNLOAD parameters and optional keywords see the discussion of LINKNLOAD in the PV-WAVE Reference.

Discussion

LINKNLOAD lets you call a C function (or a FORTRAN function) from PV-WAVE almost as if you were calling a PV-WAVE function. The called function can obtain information from PV-WAVE through passed parameters or by accessing PV-WAVE’s variables directly (see Using wavevars() to Access PV-WAVE Variables on page 50).

Any PV-WAVE data type can be passed as a parameter to a C or FORTRAN routine. By default, parameters are passed by reference (not by value), and thus it is up to the programmer’s discretion whether or not the C or FORTRAN function alters the parameter’s value. Use the Value keyword to pass parameters by value. Parameters are passed in the traditional C fashion of argc and argv. On OpenVMS only, use the Vmscall keyword to pass parameters individually rather than with argc and
argv. The C function must know the type to expect for each parameter and must cast it to a C variable of the correct type.

For FORTRAN, since the parameters are passed as pointers, functions are provided to access their values. These functions are:

- WLNL_GETBYTE
- WLNL_GETSHORT
- WLNL_GETLONG
-WLNL_GETFLOAT
- WLNL_GETDOUBLE
- WLNL_GETCOMPLEX
- WLNL_GETSTRING
- WLNL_GETBYTE_ARRAY
- WLNL_GETSHORT_ARRAY
- WLNL_GETLONG_ARRAY
- WLNL_GETFLOAT_ARRAY
- WLNL_GETDOUBLE_ARRAY
- WLNL_GETCOMPLEX_ARRAY
- WLNL_GETSTRING_ARRAY

For detailed information on these functions, see the file:

(UNIX) $WAVE_DIR/util/variables/README
(OpenVMS) WAVE_DIR:[UTIL.VARIABLES]README

NOTE Make sure the number, type, and dimension of the parameters passed to the external function match what the external function expects (this can most easily be done from within PV-WAVE before calling LINKNLOAD). Furthermore, the length of string parameters must not be altered and multi-dimensional arrays are flattened to one-dimensional arrays.

NOTE Because internal C longs are 8 bytes on Digital UNIX, passing longs from FORTRAN applications requires an INTEGER*8 declaration. Note that with LINKNLOAD, pointers are used to get the value of a variable. For example:

B = WLNL_GETBYTE(VAL(ARGP(1)))
Here, the value of ARGP represents a pointer, which is the same size as a long integer. Thus, ARGP needs to be declared as an INTEGER*8 variable.

Accessing the Data in PV-WAVE Variables

Two methods exist for accessing the results generated by PV-WAVE in a user-written application called with LINKNLOAD.

Method 1: wavevars

The first method uses wavevars, a C function that can be invoked from code linked to PV-WAVE (either statically or dynamically) to obtain data from PV-WAVE’s data variable space.

The wavevars calling sequence is:

\[
\text{result} = \text{wavevars}(&\text{argc}, &\text{argv}, &\text{argp});
\]

For detailed information about wavevars, see Using wavevars() to Access PV-WAVE Variables on page 50.

The wavevars function can only be used from a C function. It cannot be used from a FORTRAN function; however, a FORTRAN function could be used in conjunction with a C wrapper to accomplish the same task. See Example 4 on page 35 for information on this technique.

Method 2: The Option Programming Interface

The second method involves using the Option Programming Interface (OPI), a C-callable or FORTRAN-callable programming interface that was developed to provide greater flexibility and control than wavevars. OPI differs from wavevars in the following ways:

- Uses less memory than wavevars
- Can obtain information about a single PV-WAVE variable at a time.
- Can obtain a subset of the information normally returned by wavevars.

For detailed information on OPI, see the PV-WAVE Programmer’s Guide.

NOTE To use OPI effectively with C programs, you should be a C programmer, understand the difference between call-by-reference and call-by-value, and be able to use pointers and the C malloc() function. To use OPI with FORTRAN callable libraries, you should be a FORTRAN programmer, understand the difference
between call-by-reference and call-by-value, and know how to use the LOC and VAL functions. To use either the C or FORTRAN OPI library, you must know how to use the LINKNLOAD function.

Example 1: Calling a C Program

In this example, parameters are passed to the C external function using the conventional \((\text{argc}, \text{argv})\) UNIX strategy. \text{argc} indicates the number of data pointers which are passed from PV-WAVE within the array of pointers called \text{argv}. The pointers in \text{argv} can be cast to the desired type as the following program demonstrates.

You can find the following listed file in:

\$WAVE\_DIR/demo/interapp/linknload/example.c

```c
#include <stdio.h>
#include "wavevars.h"
long WaveParams(argc,argv)
   int argc;
   char *argv[];
{
   char *b;
   short *s;
   long *l;
   float *f;
   double *d;
   complex *c;
   char **str;
   if (argc != 7) {
      fprintf(stderr,"wrong # of parameters\n");
      return(0);
   }
   b = ((char **)argv)[0];
   s = ((short **)argv)[1];
   l = ((long **)argv)[2];
   f = ((float **)argv)[3];
   d = ((double **)argv)[4];
   c = ((complex **)argv)[5];
   str = ((char ***)argv)[6];
   fprintf(stderr,\"%d %d %ld %g %g <%g%gi> '%s'\n", \(\text{int}\)
   b[0], (\text{int}) s[0], (\text{long}) l[0], f[0], d[0], c[0].r, c[0].i, str[0]);
```
return(12345);
}

**Compiling the Example C Routine**

The commands for compiling the example C routine to produce a shareable object on different platforms are listed in the files:

- `$VNI_DIR/wave/demo/interapp/linknload/build.*` (UNIX)
- `VNI_DIR:[WAVE.DEMO.INTERAPP.LINKNLOAD]BUILD.*` (OpenVMS)

Please refer to the appropriate operating system documentation for more information on these commands.

**NOTE** Under AIX 3.2, the symbol entry point must be specified when the external sharable object is built, by using the “-e” flag, and thus the function symbol parameter to LINKNLOAD has no effect under AIX.

**Accessing the External Function with LINKNLOAD**

The following PV-WAVE code demonstrates how the C function defined in this example could be invoked.

```c
ln = LINKNLOAD('example_so','WaveParams', $byte(1),2,long(3), float(4),double(5), $complex(6,7),'eight')
```

The resulting output is:

1 2 3 4 5 <6,7i> ‘eight’

Using the INFO command, you can see that LINKNLOAD returns the scalar value 1.

```
INFO, ln
LN  LONG   =   1
```

The example program works with both scalars and arrays since the actual C program above only looks at the first element in the array and since PV-WAVE collapses multi-dimensional arrays to one-dimensional arrays:

```c
ln = LINKNLOAD('example_so','WaveParams', $ [byte(1)],[[2,3],[4,5]], [long(3)], $ [float(4)],[double(5)],[complex(6,7)],[‘eight’])
```

The resulting output is:

1 2 3 4 5 <6,7i> ‘eight’
Example 2: Calling a FORTRAN Program

In this example, parameters are passed from PV-WAVE to the FORTRAN external function. Variables from PV-WAVE are passed as pointers to the FORTRAN function. In the FORTRAN function, the \texttt{wlnl\_get\_} functions are used to retrieve the values of the variables. See Discussion on page 27 for more information on these functions.

You can find the following listed file in:

\texttt {$WAVE\_DIR/demo/interapp/linknload/examplefor.f$}

```
INTEGER*4 FUNCTION WAVEPARAMS(ARGC,ARGP)
     INTEGER*4 ARGC, ARGP(*)

BYTE B
INTEGER*2 S
INTEGER*4 L
REAL*4 F
DOUBLE PRECISION D
STRUCTURE /CMPLX/
   REAL R,I
END STRUCTURE
RECORD /CMPLX/ C
INTEGER*4 RET
CHARACTER*80 STR
INTEGER*4 NCHAR

INTEGER*4 WLNL_GETLONG
INTEGER*4 WLNL_GETSTRING
INTEGER*4 WLNL_GETCOMPLEX
INTEGER*2 WLNL_GETSHORT
BYTE WLNL_GETBYTE
REAL WLNL_GETFLOAT
DOUBLE PRECISION WLNL_GETDOUBLE

IF (LOC(ARGC) .NE. 7) THEN
```
PRINT *,'Wrong # of parameters',LOC(ARGC)
WAVEPARAMS = 0
RETURN
ENDORF

B = WLNL_GETBYTE(%VAL(ARGP(1)))
S = WLNL_GETSHORT(%VAL(ARGP(2)))
L = WLNL_GETLONG(%VAL(ARGP(3)))
F = WLNL_GETFLOAT(%VAL(ARGP(4)))
D = WLNL_GETDOUBLE(%VAL(ARGP(5)))
RET = WLNL_GETCOMPLEX(%VAL(ARGP(6)),C.R,C.I)
NCHAR = WLNL_GETSTRING(%VAL(ARGP(7)),STR, 80)
PRINT 100,B,S,L,F
100 FORMAT(' BYTE=',I3,',SHORT=',I6,',LONG=',I8,',REAL=',F10.5)
PRINT 200,D,C.R,C.I
200 FORMAT(' DOUBLE=',D15.5,',COMPLEX=[',F10.5,',',F10.5,']')
PRINT *,'STRING=',STR,',NCHAR=',NCHAR

WAVEPARAMS = 1
RETURN
END

**Compiling the Example FORTRAN Routine**

The commands for compiling the example FORTRAN routine to produce a shareable object on different platforms are listed in the files:

**(UNIX)**  
$VNI_DIR/wave/demo/interapp/linknload/build.*

**(OpenVMS)**  
VNI_DIR:[WAVE.DEMO.INTERAPP.LINKNLOAD]BUILD.*

Please refer to the appropriate operating system documentation for more information on these commands.

Under SunOS and Solaris the setting of LD_LIBRARY_PATH may vary depending on how your operating system was installed. If necessary, see your system administrator to determine the correct directory.
**Accessing the External Function with LINKNLOAD**

The following PV-WAVE code demonstrates how the FORTRAN function defined in this example could be invoked.

```fortran
ln = LINKNLOAD('examplefor_so','waveparams', $ 
    byte(1),2,long(3), float(4),double(5), complex(6,7),'eight')
```

The resulting output is:

1 2 3 4 5 <6,7i> 'eight'

Using the INFO command, you can see that LINKNLOAD returns the scalar value 12345 as expected.

```fortran
INFO, ln
    LN LONG = 12345
```

The example program works with both scalars and arrays since the actual FORTRAN program above only looks at the first element in the array and since PV-WAVE collapses multi-dimensional arrays to one-dimensional arrays:

```fortran
ln = LINKNLOAD('examplefor_so','waveparams', $ 
    [byte(1)],[[2,3],[4,5]], [long(3)], $ 
    [float(4)],[double(5)],[complex(6,7)], ['eight'])
```

The resulting output is:

1 2 3 4 5 <6,7i> 'eight'

**Example 3**

In this example, the PV-WAVE program calls the C function (via LINKNLOAD), passing it parameters. The C function modifies these parameters and also accesses PV-WAVE’s variable data space directly. The C function then returns control to PV-WAVE, passing to PV-WAVE the result of the function.

This example contains two programs:

- `wave_to_c_main.pro` — A PV-WAVE main program that uses LINKNLOAD to call a C function. The PV-WAVE program passes parameters to the C function, which modifies the parameters’ values. When control is returned to PV-WAVE, the values of the PV-WAVE variables which were passed as parameters are changed.

- `c_from_wave.c` — The C function to be called by PV-WAVE via LINKNLOAD.

On a UNIX system, the C and PV-WAVE code described in this example is available online in the directory:
On an OpenVMS system, the C and PV-WAVE code described in this example is available online in the directory:

WAVE_DIR: [DEMO.INTERAPP.LINKNLOAD]

The C function must be compiled as a shareable object, as explained below. It is because the C function is linked shareable and shares the same data space with PV-WAVE that the C function can access PV-WAVE variables directly.

**Compiling the Example C Routine**

The commands for compiling the example C routine to produce a shareable object on different platforms are listed in the files:

(UNIX)  $VNI_DIR/wave/demo/interapp/linknload/build.*

(OpenVMS)  VNI_DIR: [WAVE.DEMO.INTERAPP.LINKNLOAD] BUILD.*

Please refer to the appropriate operating system documentation for more information on these commands.

**Running the Example Program**

Enter PV-WAVE and type the following at the WAVE> prompt:

.RUN wave_to_c_main

Under Solaris, you need to do set the LD_LIBRARY_PATH environment variable to the directory containing the file c_from_wave.so. You must set this variable even if that file is in the current working directory. For example:

solaris% setenv LD_LIBRARY_PATH .

**Example 4**

In this example, the PV-WAVE program calls a C wrapper function, passing it PV-WAVE variables as parameters. The C wrapper calls the FORTRAN function, passing along the same parameters. The FORTRAN function modifies the values of the PV-WAVE variables it receives as parameters. The FORTRAN function then returns control to PV-WAVE, passing to PV-WAVE the result of the function. The new values, assigned to the PV-WAVE variables by the FORTRAN program, are accessible within PV-WAVE upon return from the FORTRAN program.

This example contains three programs:

- wave_to_fort_main.pro — A PV-WAVE main program that uses LINKNLOAD to call a FORTRAN function. The PV-WAVE program passes
parameters to a C wrapper, which in turn calls the desired FORTRAN function to modify the PV-WAVE parameters. When control is returned to PV-WAVE, the values of the PV-WAVE variables which were passed as parameters are changed.

- **wave_to_fort_w.c** — A C function that is called by PV-WAVE, via LINKNLOAD. This routine is a C wrapper that allows a PV-WAVE program to pass parameters to and invoke a FORTRAN function. The FORTRAN function can modify the values of the PV-WAVE variables passed as parameters and the modified values will be accurately reflected upon return to PV-WAVE.

- **fort_from_wave.f** — The FORTRAN function to be called by the C wrapper.

On a UNIX system, the C and PV-WAVE code described in this example is available online in the directory:

```
$WAVE_DIR/demo/interapp/linknload
```

On an OpenVMS system, the C and PV-WAVE code described in this example is available online in the directory:

```
WAVE_DIR: [DEMO.INTERAPP.LINKNLOAD]
```

### Compiling and Linking the Programs

The commands for compiling the example C wrapper function `wave_to_fort_w` and the FORTRAN function `fort_from_wave.f` to produce a shareable object on different platforms are listed in the files:

(UNIX) `SVNI_DIR/wave/demo/interapp/linknload/build.*`

(OpenVMS) `VNI_DIR: [WAVE.DEMO.INTERAPP.LINKNLOAD]BUILD.*`

Please refer to the appropriate operating system documentation for more information on these commands.

Under SunOS the setting of LD_LIBRARY_PATH may vary depending on how your operating system was installed. If necessary, see your system administrator to determine the correct directory.

### Running the Example Program

Enter PV-WAVE and type the following at the `WAVE>` prompt:

```
.RUN wave_to_fort_main
```

Under Solaris, you need to set the LD_LIBRARY_PATH environment variable to the directory containing the file `wave_to_fort.so`. You must set this variable even if that file is in the current working directory.
Calling PV-WAVE in a Statically Linked Program

Under OpenVMS or UNIX, an application written in C or FORTRAN can be linked directly (statically) with the PV-WAVE object libraries. The user application then passes PV-WAVE commands to the entry points cwavec (C application) or cwavefor (FORTRAN application) in the PV-WAVE shareable image.

cwavec: Calling PV-WAVE from a C Program

The routine cwavec, discussed in detail in this section, is the C application entry point to a PV-WAVE shareable image.

Usage

\[
\text{istat} = \text{cwavec}(\text{action}, \text{numcmds}, \text{cmds})
\]

Parameters

\text{action} — Specifies how you wish PV-WAVE to execute. It can have one of the following values:

- \text{action}=1 — Run normally. You are interactively prompted for input and execution continues until you enter the end-of-file character or issue the EXIT command. At this point, cwavec returns with a value of 1. Once cwavec has been called in this mode, it should not be called again.
- \text{action}=2 — Execute the commands supplied by \text{cmds} array and return. The return value is the value of the !Error system variable. The cwavec routine can be called repeatedly in this mode.
- \text{action}=3 — It is necessary to wrap up the session by calling cwavec one last time with \text{action}=3. This performs any housekeeping required by PV-WAVE such as closing any open files. The return value for this mode is 1. Once cwavec has been called in this mode, it should not be called again.

\text{numcmds} — The number of elements supplied in \text{cmds}. This argument is ignored if \text{action}=3 or if \text{action}=1.

\text{cmds} — An array of pointers to strings. If \text{action}=2, \text{cmds} provides an array of PV-WAVE commands to execute. This argument is ignored if \text{action}=3 or if \text{action}=1.

Returned Value

\text{istat} — The returned value depends on the \text{action} selected, as explained previously.
**Discussion**

You can choose to communicate with PV-WAVE in either an interactive mode or by sending an array of commands. Both of these methods automatically initialize PV-WAVE.

The first parameter is the *action* parameter. The action parameter may have one of the following the values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Run PV-WAVE interactively.</td>
</tr>
<tr>
<td>2</td>
<td>Execute a sequence of PV-WAVE commands and return to the C program.</td>
</tr>
<tr>
<td>3</td>
<td>Exit PV-WAVE and return to the C program.</td>
</tr>
</tbody>
</table>

The third parameter is the name of an array of pointers to strings (i.e., `char**`) containing the PV-WAVE commands to be executed. The second parameter specifies the number of elements supplied in the third parameter. The second and third parameters are ignored if the value of the *action* parameter is 1 or 3.

The status value returned by `cwavec` depends on the value of the *action* parameter and in some cases on the value of the action performed. If the value of the *action* parameter is 1 or 3, `cwavec` will return 1 as the status. If the value of the *action* parameter is 2, `cwavec` will return the value of the PV-WAVE system variable !Err as the status.

**Accessing the Data in PV-WAVE Variables**

To access data in PV-WAVE variables, use the `wavevars` function, a C function that can be invoked from code linked to PV-WAVE (either statically or dynamically).

The `wavevars` calling sequence is:

```
result = wavevars(&argc, &argv, &argp);
```

For detailed information about `wavevars`, see *Using wavevars() to Access PV-WAVE Variables* on page 50.
Ending the Session with PV-WAVE

If you are in interactive mode (action=1), enter EXIT at the WAVE> prompt to return to your C application. There is no need to call cwavec with action=3 to end the session. However, if the application has accessed PV-WAVE in non-interactive mode (action=2), the session must be terminated by a final call to cwavec with action=3.

Running PV-WAVE from a C Program

To run PV-WAVE from a C program you must first link the C program with PV-WAVE. The C program may then invoke PV-WAVE via the entry point cwavec in the PV-WAVE shareable object. The C program must pass three parameters to the cwavec entry point. For details on linking the application to PV-WAVE, see How to Link Applications to PV-WAVE on page 47.

Example 1

In non-interactive mode, valid PV-WAVE commands are passed to cwavec as an array of strings. For example, to plot the vector \([1, 2, 3, 4, 5]\) from a C application statically linked to PV-WAVE, the commands would be:

```c
char *cmds[5];
.
.
cmds[0] = "a = indgen(5) + 1";
cmds[1] = "plot, a";
action=2;
status = cwavec(action, 2, cmds);
```

Example 2

This example shows how to pass a five-element array to PV-WAVE via cwavec, have PV-WAVE perform some calculations, and produce a plot.

You can find the following listed file in:

$WAVE_DIR/demo/interapp/cwave/example.c

```c
#include <stdio.h>
main()
{
    /* Variables for array calculations */
    */
```c
int action, numcmds, istat, cwavec();
char *cmds[5];
/*
 * Access PV-WAVE in non-interactive mode
 */
action = 2;
numcmds = 5;
/*
 * Send the array of commands to PV-WAVE
 * Define the array A
 * Perform matrix multiplication
 * Print contents of B
 * Display B as a surface
 * Issue a wait command so you can view result
 * Call cwavec
 */
cmds[0] = "A = INDGEN(5) * 4";
cmds[1] = "B = A # A";
cmds[2] = "PRINT, B";
cmds[3] = "SURFACE, B";
cmds[4] = "WAIT, 3.0";
istik = cwavec(action, numcmds, cmds);
/*
 * Since we are done sending commands to
 * PV-WAVE, make a final call to cwavec
 * with action=3 to wrap up the session
 */
action = 3;
istik = cwavec(action, 0, cmds);
}

Compiling and Linking the Example Program

You can use the following commands to compile the example program and link it
to PV-WAVE on a UNIX system:

```bash
setenv arch '$/WAVE_DIR/bin/arch'
cc -c example.c
make -f $WAVE_DIR/src/pub/quick.mk link MAIN=example OBJ=example.o TARGARCH=$arch
```
Example 3

In this example the C program passes commands to PV-WAVE to be executed and then accesses the results directly from PV-WAVE’s variable data space via the wavevars routine.

This example uses one program:

- wave_from_c.c — The C function that calls PV-WAVE and accesses PV-WAVE variables directly.

On a UNIX system, this program is available online in the directory:

$WAVE_DIR/demo/interapp/cwave

On an OpenVMS system, this program is available online in the directory:

WAVE_DIR:\[DEMO.INTERAPP.CWAVE\]

The C program must be compiled and linked with PV-WAVE to produce a single executable program, as explained in the next section. It is because your program is linked with PV-WAVE as a single executable that your program can “share” PV-WAVE variables.

Compiling and Linking the Example Program

You can use the following commands to compile the example program and link it to PV-WAVE on a UNIX system:

```
setenv arch '$WAVE_DIR/bin/arch'
cc -c wave_from_c.c
make -f $WAVE_DIR/src/pub/quick.mk \
    link MAIN=wave_from_c OBJ=wave_from_c.o \
    TARGARCH=$arch
```

For more information on compiling programs and linking them using the makefile quick.mk, see *How to Link Applications to PV-WAVE* on page 47.

**NOTE** The link operation for this example creates a large executable, because it links in all of PV-WAVE.
Running the Program

After the program is compiled and linked, it can be run by entering the name of the resulting executable file. For example, if the executable is called `wave_from_c`, enter:

```
wave_from_c
```

The output from this example is shown in Figure 2-1.

![Figure 2-1](image)

The first graphic produced by this example is shown on the left. The second graphic produced is on the right.

cwavefor: Calling PV-WAVE from a FORTRAN Program

The `cwavefor` routine is the FORTRAN application entry point to a PV-WAVE shareable image.

**Usage**

```
istat = cwavefor(action, numcmds, cmds, cmdlen)
```

**Parameters**

- `action` — Specifies how you wish PV-WAVE to execute. It can have one of the following values:
  - `action=1` — Run normally. You are interactively prompted for input and execution continues until you enter the end-of-file character or issue the EXIT command. At this point, `cwavefor` returns with a value of 1. Once `cwavefor` has been called in this mode, it should not be called again.
• *action*=2 — Execute the commands supplied by *cmds* array and return. The return value is the value of the !Error system variable. The *cwavefor* routine can be called repeatedly in this mode.

• *action*=3 — It is necessary to wrap up the session by calling *cwavefor* one last time with *action*=3. This performs any housekeeping required by PV-WAVE such as closing any open files. The return value for this mode is 1. Once *cwavefor* has been called in this mode, it should not be called again.

**numcmds** — The number of elements supplied in *cmds*. This argument is ignored if *action*=3 or if *action*=1.

**cmds** — An array of strings. If *action*=2, *cmds* provides an array of PV-WAVE commands to execute. This argument is ignored if *action*=3 or if *action*=1.

**cmdlen** — The declared length of each string element in the two-dimensional array.

**Returned Value**

**istat** — The returned value depends on the *action* selected, as explained previously.

**Discussion**

You can choose to communicate with PV-WAVE in either an interactive mode or by sending an array of commands. These methods automatically initialize PV-WAVE.

The first parameter is the *action* parameter. The action parameter may have one of the following the values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Run PV-WAVE interactively.</td>
</tr>
<tr>
<td>2</td>
<td>Execute a sequence of PV-WAVE commands and return to the FORTRAN program.</td>
</tr>
<tr>
<td>3</td>
<td>Exit PV-WAVE and return to the FORTRAN program.</td>
</tr>
</tbody>
</table>

The third parameter is the name of an array of strings containing the PV-WAVE commands to be executed. The second parameter specifies the number of elements supplied in the third parameter. The second and third parameters are ignored if the value of the *action* parameter is 1 or 3.

The status value returned by *cwavefor* depends on the value of the *action* parameter and in some cases on the value of the action performed. If the value of the
The parameter is 1 or 3, `cwavefor` will return 1 as the status. If the value of the `action` parameter is 2, `cwavefor` will return the value of the PV-WAVE system variable !Error as the status.

**Ending the Session with PV-WAVE**

If you are in interactive mode ([action=1](#)), enter EXIT at the WAVE> prompt to return to your FORTRAN application. There is no need to call `cwavefor` with [action=3](#) to end the session. However, if the application has accessed PV-WAVE in non-interactive mode ([action=2](#)), the session must be terminated by a final call to `cwavefor` with [action=3](#).

**Running PV-WAVE from a FORTRAN Program**

To run PV-WAVE from a FORTRAN program you must first link the FORTRAN program with PV-WAVE. The FORTRAN program can then invoke PV-WAVE via the entry point `cwavefor` in the PV-WAVE shareable object. The FORTRAN program must pass four parameters to the `cwavefor` entry point. For details on linking the application to PV-WAVE, see *How to Link Applications to PV-WAVE* on page 47.

**Example 1**

In non-interactive mode, valid PV-WAVE commands are passed to `cwavefor` as an array of strings. For example, to plot the vector [1, 2, 3, 4, 5] from a FORTRAN application statically linked to PV-WAVE, the commands would be:

```fortran
character *50 cmds(5)
.
.
cmds(1) = 'a = INDGEN(5) + 1'
cmds(2) = 'plot, a'
action=2
call cwavefor(action, 2, cmds, 50)
```

**Example 2**

This example shows how to pass a five-element array to PV-WAVE via `cwavefor`, have PV-WAVE perform some calculations, and produce a plot. You can find the following listed file in:

- `$WAVE_DIR/demo/interapp/cwave/examplefor.f`

```fortran
PROGRAM EXAMPLE_175
```
C
C Variables for array calculations
C
  integer*4  action, numcmds, istat, cwavefor
  character *30  cmds(5)
C
C In order to initialize stdin and stdout
C correctly so that output goes to your
C terminal, make the following call:
C This call for OpenVMS only.
  call vaxc$crtl_init
C
C Access PV-WAVE in non-interactive mode
C
  action=2
  numcmds = 5
C
C Send the array of commands to PV-WAVE
C Define the array A
C Perform matrix multiplication
C Print contents of B
C Display B as a surface
C Issue a wait command so user can view result
C Call cwavefor
C
  cmds(1) = 'A = INDGEN(5) *4'
  cmds(2) = 'B = A # A'
  cmds(3) = 'PRINT, B'
  cmds(4) = 'SURFACE, B'
  cmds(5) = 'WAIT, 3.0'
  istat = cwavefor(action, numcmds, cmds, 30)
C
C Since we are done sending commands to
C PV-WAVE, make a final call to cwavec
C with action=3 to wrap up the session.
C
  action=3
  istat = cwavefor(action, 0, cmds, 30)
  stop
end
Compiling and Linking the Example Program

You can use the following commands to compile the example program and link it to PV-WAVE on a UNIX system:

```bash
setenv arch `$WAVE_DIR/bin/arch'
f77 -c examplefor.f
make -f $WAVE_DIR/src/pub/quick.mk 
   flink MAIN=examplefor OBJ=examplefor.o 
   TARGARCH=$arch
```

For more information on compiling programs and linking them using the makefile `quick.mk`, see *How to Link Applications to PV-WAVE* on page 47.

Example 3

In this example, the FORTRAN program passes commands to PV-WAVE to be executed and then accesses the results directly (via a C wrapper) from PV-WAVE’s variable data space using the C function `wavevars`.

This example uses two functions:
- `wave_from_fort.f` — The FORTRAN function that calls PV-WAVE and accesses PV-WAVE variables directly.
- `wavevars_f1.c` — A C function (wrapper) that allows the FORTRAN program to retrieve and/or modify the values of floating-point arrays in PV-WAVE’s variable data space. This is accomplished via the `wavevars` function, which interacts directly with PV-WAVE’s variable data space. (Direct interaction between a FORTRAN program and `wavevars` does not work because FORTRAN lacks the C language’s ability to access a common data area by address.)

On a UNIX system, the C and FORTRAN code described in this example is available online in the directory:

```
$WAVE_DIR/demo/interapp/cwave
```

On an OpenVMS system, the C and FORTRAN code described in this example is available online in the directory:

```
WAVE_DIR:[DEMO.INTERAPP.CWAVE]
```

The FORTRAN program must be compiled and linked with PV-WAVE and the C wrapper routine to produce a single executable program, as explained in the next section. It is because your program is linked with PV-WAVE as a single executable that your program can “share” PV-WAVE variables.
Compiling and Linking the Example Program

You can use the following commands to compile the example program and link it to PV-WAVE on a UNIX system:

```bash
setenv arch '$WAVE_DIR/bin/arch'
f77 -c wave_from_fort.f
cc -c wavevars_fl.c
make -f $WAVE_DIR/src/pub/quick.mk \
   flink MAIN=wave_from_fort \
   OBJ="wave_from_fort.o wavevars_f1.o" \
   TARGARCH=$arch
```

For more information on compiling programs and linking them using the makefile `quick.mk`, see *How to Link Applications to PV-WAVE* on page 47.

**NOTE** The link operation for this example creates a large executable, because it links in all of PV-WAVE.

Running the Program

After the program is compiled and linked, it can be run by entering the name of the resulting executable file. For example if the executable is called `wave_from_fort`, enter:

```bash
wave_from_fort
```

The output from this example is shown in *Figure 2-1* on page 42.

How to Link Applications to PV-WAVE

This section explains how to link C and FORTRAN applications to PV-WAVE on a UNIX workstation and how to link C applications to PV-WAVE under OpenVMS.

**NOTE** To relink PV-WAVE on Solaris systems, you must use the SPARCworks C compiler. PV-WAVE references a number of routines in the SPARCworks math library that are not available on other compilers. Visual Numerics will consider providing support for other compilers if there is sufficient demand. Please contact Visual Numerics Technical Support to request support for a different compiler.
Using the quick.mk Makefile on a UNIX System

The makefile quick.mk is provided to assist you in linking compiled C and FORTRAN programs to PV-WAVE.

NOTE On some systems, before invoking the makefile quick.mk with the wave, link, or flink targets, you may need to run ranlib on the PV-WAVE archives using the ranlib target supplied in quick.mk.

On a UNIX system, this makefile is located in:

$WAVE_DIR/src/pub/quick.mk

For example, to link a compiled program called mywavec to PV-WAVE, you can use the following commands:

host> setenv arch '$WAVE_DIR/bin/arch'
host> make -f $WAVE_DIR/src/pub/quick.mk \
      link MAIN=mywavec OBJ=mywavec.o \ 
      TARGARCH=$arch

Note the backquotes must be entered exactly as shown.

quick.mk can be modified as desired to build a single PV-WAVE application or multiple applications. Refer to $WAVE_DIR/src/pub/README for more details.

Linking a C Application to PV-WAVE: UNIX

The following statements illustrate how to link an application written in C to PV-WAVE on a Sun-4 (UNIX) system. Note that the quotes and backquotes must be entered exactly as shown.

NOTE The following commands are shown only as an example. It is possible that these link commands differ from the commands required to link an application under the current release of PV-WAVE. It is recommended that you use the makefile quick.mk, described in the previous section, to link applications to PV-WAVE.

setenv arch '$WAVE_DIR/bin/arch'
setenv BINARCH $WAVE_DIR/bin/bin.$arch
setenv ARCHLIB \
   '-Bstatic -lx11 -Bdynamic -ltermcap -lm -ldl $' \
   '-ltermcap -lm'
Linking a FORTRAN Application: UNIX

The following statements illustrate how to link an application written in FORTRAN to PV-WAVE on a Sun-4 (UNIX) system. The backquotes must be entered exactly as shown.

The following commands are shown only as an example. It is possible that these link commands differ from the commands required to link an application under the current release of PV-WAVE. It is recommended that you use the makefile quick.mk, described in the previous section, to link applications to PV-WAVE.

```
setenv arch `$WAVE_DIR/bin/arch`
setenv BINARCH `$WAVE_DIR/bin/bin.$arch`
setenv ARCHLIB \`
    -Bstatic -lX11 -Bdynamic -ltermcap -lm -ldl $`
                   \`
    -ltermcap -lm'

f77 -Bstatic -o mycwavefor mycwavefor.o \`
    $BINARCH/wave.$arch.a \`
    $VNI_DIR/cmathstat/bin/bin.$arch/cmast.$arch.a \`
    $VNI_DIR/dblink/bin/bin.$arch/dbms.$arch.a \`
    $BINARCH/dc.$arch.a \`
    $BINARCH/nr.$arch.a \`
    $BINARCH/render.$arch.a \`
    $BINARCH/rpc.$arch.a \`
    $BINARCH/table.$arch.a \`
    $BINARCH/wt.$arch.a \`
    $BINARCH/optionstubs.$arch.a \`
    $BINARCH/wave.$arch.a \`
    $ARCHLIB
```
Linking a C Application: OpenVMS

The following DCL procedure illustrates how to link an application written in C to PV-WAVE on an OpenVMS system:

```
$! CLINK.COM --- link a C program with
$! PV-WAVE using the X driver
$!
$  cc my_c_app.c
$  define wave_image wave_dir:[bin]wave_x.exe
$  link my_c_app, sys$input/option/share
    wave_image/shareable
$!
```

Refer to the file: WAVE_DIR:[SRC.PUB]README.VVMS for more information on linking programs with PV-WAVE on an OpenVMS system.

Using wavevars() to Access PV-WAVE Variables

You can access PV-WAVE variables from a C program by calling the function wavevars. Once commands have been sent to PV-WAVE from an external application, you can use the wavevars function to access the results in the external application. wavevars is a C function that can be invoked from code linked to PV-WAVE either:

- statically via `cwavec`, or
- dynamically via `LINKNLOAD`.

**NOTE**  Direct interaction between a FORTRAN program and wavevars is not possible because FORTRAN lacks C’s ability to access a common data area by address. Thus, to access PV-WAVE variables from a FORTRAN program, a C wrapper must be written that calls wavevars.

wavevars obtains data directly from PV-WAVE’s variable data space.

**NOTE**  See also the section Method 2: The Option Programming Interface on page 29. The Option Programming Interface (OPI) functions allow user-written code to access PV-WAVE variables and use other PV-WAVE functionality. OPI provides greater flexibility and control than wavevars.
Usage

```c
int argc;
char **argv;
WaveVariable *argp;
result = wavevars(&argc, &argv, &argp);
```

Parameters

**argc** — Set to the number of variables returned.

**argv** — Set to be an array of strings, sorted in lexicographic order, corresponding to variable names available at the current scope level of PV-WAVE.

**argp** — A type WaveVariable array of descriptors defining the type, structure, and dimension of the variables as well as providing a pointer to their actual data. The WaveVariable structure is described in the Discussion section that follows.

Returned Value

**result** — A C int value which is nonzero if the routine executed successfully, and zero if an error (such as running out of memory) occurred.

Discussion

PV-WAVE variables can be accessed directly from a C function by calling the C function **wavevars** which is linked into PV-WAVE. The C function passes three parameters to the wavevars entry point.

The first parameter is the address of an integer variable into which wavevars will return the number of currently-defined PV-WAVE variables (including system variables).

The second parameter is the address of an array of pointers to strings (i.e., char**) into which wavevars will return the names of currently-defined PV-WAVE variables.

The third parameter is the address of an array of pointers to the C structure WaveVariable into which wavevars will return information regarding the type, structure, dimension, and data of each PV-WAVE variable (including a pointer to the current value of the variable).

WaveVariable is defined as follows in $WAVE_DIR/util/variables/wavevars.h. This header file must be included in any C function that calls wavevars.
typedef struct WaveVariable {
    int type;
    int read_only;
    int numdims;
    int dims[8];
    int numelems;
    int *data;
    char name[MAXIDLEN + 1];
} WaveVariable;

**CAUTION** Although *wavevars* returns pointers to the data associated with PV-WAVE’s variables, keep in mind that the data pointer associated with a given variable can change after execution of certain PV-WAVE system commands. It’s best to call *wavevars* immediately before it is needed to obtain information from the external program.

The *wavevars* function allocates space to store the information it returns to the caller. When the caller no longer needs the information returned by *wavevars*, then the *free_wavevars()* function should be called to free the space. The arguments to *free_wavevars()* should be identical to those used in the call to *wavevars* such as:

```
result = free_wavevars( &argc, &argv, &argp );
```

and *argc* must still contain the number of variables returned by the *wavevars* call.

The WaveVariable structure’s fields are:

**int type** — The *type* field indicates the type of the variable. Valid PV-WAVE variable types, together with their C equivalents, are defined in *wavevars.h* as follows:

- TYP_BYTE char;
- TYP_INT short;
- TYP_LONG long;
- TYP_FLOAT float;
- TYP_DOUBLE double;
- TYP_COMPLEX struct { float r, i; } COMPLEX;
- TYP_STRING char *

In PV-WAVE, a structure is a collection of data where each field (tag) has a name. The C structure *WaveVariable* describes a PV-WAVE structure with a type of TYP_STRUCT, where each element of the structure is contained in a list of
WaveVariable structures pointed to by the data field, which is described later in this section.

The constant TYP_ARRAY will be bitwise or-ed into the type field if the variable is in fact an array.

**int read_only** — Many PV-WAVE variables are read-only, and thus if this field is nonzero, it is not permissible to alter the actual variable data. This is often the case with system variables.

**int numdims** — PV-WAVE variables may be of dimension zero (scalar) to eight. The field numdims indicates the dimensionality of the variable.

**int dims[8]** — Indicates the size of each dimension of a variable if it is of type array.

**int numelems** — Corresponds to the total number of data values which are addressable from the data pointer.

**int *data** — Corresponds to the address of the actual variable data. The data is always stored as a one-dimensional C array regardless of the dimensionality of the PV-WAVE variable.

**char name[MAXIDLEN + 1]** — Only used when the variable being described is of type structure and represents the structure or tag field name (depending on context).

To access a specific PV-WAVE variable you must search the array of variable names returned by wavevars to find the index associated with that variable. Then use the index to access the correct PV-WAVE variable from the WaveVariable array. The type field in WaveVariable is used to determine a variable’s type. To access the data associated with a PV-WAVE variable it is necessary to use the data pointer and cast it to the correct type. It is then possible to read and/or modify the actual data value(s).

**Example 1**

The following is a simple program that retrieves a list of all PV-WAVE variables and prints out their contents. The code fragment demonstrates several important concepts.

- The data pointer must be cast to appropriate type.
- The data is always stored as a flat one-dimensional array.
- PV-WAVE structures are stored recursively.

You can find the following listed file in:
$\texttt{WAVE_DIR/demo/interapp/wavevars/example.c}$

```c
#include <stdio.h>
#include "wavevars.h"

printallvars() /* display names & value of all WAVE variables */
{
    int nvars, i;
    char **names;
    WaveVariable *vars;

    if (wavevars(&nvars,&names,&vars)) {
        for (i=0; i<nvars; i++) {
            fprintf(stderr, "%s\n",names[i]);
            printvar(& (vars[i]) );
            fprintf(stderr, "\n");
        }
        free_wavevars(&nvars, &names, &vars);
    }
}

printvar(v) /* print a WAVE variable on stderr */
WaveVariable *v;
{
    if ( v->name )
        fprintf(stderr, "tag: '%s'\n",v->name);
    else
        fprintf(stderr, "tag: \n");
    if (v->read_only)
        fprintf(stderr, "stat: READ_ONLY\n");
    else
        fprintf(stderr, "stat: READ/WRIT\n");
    fprintf(stderr, "\n\tnelems: %d\n",v->numelems);
    fprintf(stderr, "\tn_dims: %d\n",v->numdims);
    fprintf(stderr, "\tdims:%d %d %d %d %d %d %d %d\n",
            v->dims[0],v->dims[1],v->dims[2],
            v->dims[3],
            v->dims[4],v->dims[5],v->dims[6],
            v->dims[7]);
    printdata(v,v->numelems);
}
```

---

**Application Developer's Guide**
printdata(v, len)
WaveVariable *v;
int len;
{
    int i;

    if (v->type & TYP_ARRAY)
        fprintf(stderr, "\ttype: ARRAY OF ");
    else
        fprintf(stderr, "\ttype: ");

    switch(v->type & ~TYP_ARRAY) {
        case TYP_BYTE:
            {
                char *b = ((char *)v->data);
                fprintf(stderr, "BYTE\n");
                fprintf(stderr, "\tdata: ");
                for (i=0; i<len; i++)
                    fprintf(stderr, "%d ",(int)b[i]);
            }
            break;
        case TYP_INT:
            {
                short *b = ((short *)v->data);
                fprintf(stderr, "INTEGER\n");
                fprintf(stderr, "\tdata: ");
                for (i=0; i<len; i++)
                    fprintf(stderr, "%d ",(int)b[i]);
            }
            break;
        case TYP_LONG:
            {
                long *b = ((long *)v->data);
                fprintf(stderr, "LONG\n");
                fprintf(stderr, "\tdata: ");
                for (i=0; i<len; i++)
                    fprintf(stderr, "%ld ",b[i]);
            }
            break;
        case TYP_FLOAT:
            {

        }
float *b = ((float *)v->data);
fprintf(stderr,"FLOAT\n");
fprintf(stderr, "\tdata: ");
for (i=0; i<len; i++)
    fprintf(stderr, "%g ",b[i]);
}
break;
case TYP_DOUBLE:
{
    double *b = ((double *)v->data);
    fprintf(stderr,"DOUBLE\n");
    fprintf(stderr, "\tdata: ");
    for (i=0; i<len; i++)
        fprintf(stderr, "%g ",b[i]);
}
break;
case TYP_COMPLEX:
{
    Complex *b = ((Complex *)v->data);
    fprintf(stderr,"COMPLEX\n");
    fprintf(stderr, "\tdata: ");
    for (i=0; i<len; i++)
        fprintf(stderr, "<%g,%g> ",b[i].r, b[i].i);
}
break;
case TYP_STRING:
{
    char **b = ((char **)v->data);
    fprintf(stderr,"STRING\n");
    fprintf(stderr, "\tdata: ");
    for (i=0; i<len; i++)
        if ( b[i] )
            fprintf(stderr, "\'%s\' ",b[i]);
}
break;
case TYP_STRUCT:
{
    WaveVariable *b= ((WaveVariable *)
v->data);
    fprintf(stderr,"STRUCTURE\n");
    fprintf(stderr, "\tdata: ");
    for (i=0; i<len; i++)
    {
fprintf(stderr, "\n");
printvar(&(b[i]));
}
}
break;

Example 2
For an example that shows how to call PV-WAVE and access its data space from a C program, see *Example 3* on page 41.

Example 3
For an example that shows how to call PV-WAVE and access its data space from a FORTRAN program, see *Example 3* on page 46.

Example 4
For an example that shows how to call a C program from within PV-WAVE and have the C program access PV-WAVE’s variable data space, see *Example 3* on page 34.

Example 5
For an example that shows how to call a FORTRAN program from within PV-WAVE and have the FORTRAN program access PV-WAVE’s variable data space (via a C wrapper), see *Example 4* on page 35.

**Special Considerations for Noninteractive Applications**

If you are writing a noninteractive application — one for which no input is required at the PV-WAVE prompt — then you may need to build some special handling into your code for any nonblocking PV-WAVE widgets. You may also need to make special arrangements for graphic-window resizing.

**Using Nonblocking PV-WAVE Widgets**

When PV-WAVE’s input is not coming from an interactive terminal, then the non-blocking Widget event loop (`WWLoop`, `/NoBlock`, for example) may not behave as expected. This event loop is normally serviced as PV-WAVE “watches” the key-
board. Thus, when you do not have an interactive terminal, the event loop is not serviced.

Some examples of noninteractive applications include:

- Running commands from within a PV-WAVE procedure or batch file using @ or .RUN
- Sending commands to PV-WAVE via a pipe using wavecmd()
- Running PV-WAVE in batch mode or redirecting standard input
- Using cwavec() or cwavefor() (see the Note, below)

In these situations, it becomes the programmer’s responsibility to service the event loop. If you are running commands in a procedure file, then this can be as simple as exiting back to the command prompt.

If you are running PV-WAVE as a background process, however, or in some mode where you need a nonblocking event loop but do not have an interactive terminal, then you will need to call WtProcessEvent periodically to service the event loop.

To process all pending events, you call this function using the /Drain keyword.

Example

Say you have the following in a procedure file:

```
NAVIGATOR
 ; A non-blocking Widget application
WHILE ( WtProcessEvent(/Drain) NE 1 ) DO BEGIN $
   PRINT, ‘Still alive’ & $
   WAIT, 0.1
PRINT, ‘Exiting’
EXIT
```

When you execute this procedure at the PV-WAVE prompt, the WHILE loop causes WtProcessEvent to be called until the widget application is exited. Of course, in a real case, you would do some other processing rather than just PRINT and WAIT, but the point is that you need to call WtProcessEvent periodically and check its return status (so you know when the Widget application is finished).

NOTE When you use cwavec() and cwavefor(), the normal behavior for WwLoop, /NoBlock is to ignore the keyword and actually block. So if you call cwavec() to execute a nonblocking Widget, PV-WAVE will service its own event loop until the Widget exits. If you wish to have the event loop nonblocking, then
you must call WtProcessEvent periodically to process the events and use 
WwLoop, NoBlock=2 to force the nonblocking behavior.

**Window Resizing**

PV-WAVE will not automatically recognize resizing of graphic windows when it is not being run interactively (for example, using wavecmd() or cwavec()). To notify PV-WAVE of window resizing use:

```
WSET, winid, /Resize
```

This command has the effect of updating the !D system variable. See WSET for more information.

---

**Communication with Remote Procedure Calls (UNIX Only)**

PV-WAVE can communicate with user-written applications by establishing a client/server relationship based upon Sun Remote Procedure Call (RPC) technology. PV-WAVE is able to act as either the client or server depending upon the requirements of the particular user-written application. This means that the user-written application can also act as either the client or server.

**Remote Procedure Call (RPC) Technology**

Remote procedure calls are a high-level communications paradigm which allow distributed applications to be developed using procedure calls which shield the programmer from knowing the details of the underlying network mechanisms. RPC implements a client-to-server communications system designed to support interapplication communication over a network. Data for RPC calls is transmitted using the External Data Representation (XDR) Standard.

PV-WAVE uses RPC calls to transmit data in XDR format between hosts across the network or between processes on the same host. The parameters used with the functions CALL_UNIX or CALL_WAVE (described later in this section) are “packed” into a proprietary variable structure, transmitted in an XDR format and then “unpacked” and made available for PV-WAVE and/or the user’s application.

For the purpose of our discussion, the following terms and definitions are used. A client initiates the remote procedure calls to a server. A server provides network access to a collection of one or more remote applications. A server may support multiple remote applications or multiple versions of the same remote application.
A client initiates the interapplication communication by sending a procedure call message to the server process and then waits (blocks) for a procedure reply message. The procedure call message contains the remote application’s parameters, among other things. The procedure reply message contains the results from the remote application, among other things. Once the client has received the results through the procedure reply message, the results are made available to the client and program execution continues.

On the server side, the process is dormant while it awaits the arrival of a procedure call message. When this message arrives, the server process extracts the parameters passed from the client and passes them to the remote application. The remote application performs its tasks with the supplied parameters and returns the results to the server process. The server then sends a procedure reply message to the client and returns to a dormant state awaiting the next procedure call message from a client.

**Synchronization of Client and Server Processes**

Because client and server processes can be started and run independently of one another, it is possible for the client and server processes to be out of synchronization. In general, especially on relatively unloaded systems, synchronization is not a problem. However, it is good practice to ensure that the server process is in a state that can handle client requests before the client process is started.

If PV-WAVE is the server process, then it should be in a state to begin receiving client requests after the following message appears (assuming that the !Quiet system variable is zero):

```
Compiled - UNIX_LISTEN
```

If an external program that you wrote is used as the server process, you might want to include code in the program that lets PV-WAVE clients know when it is appropriate to begin RPC transactions.

Another alternative is simply to wait for a period of time before starting the client process; however, the amount of time required will vary greatly depending on the load of the client system and the load of the server system, which in some cases may be the same system.

**Linking a Server or a Client with PV-WAVE**

The PV-WAVE archive library contains all the routines used for RPC-based interapplication communication.
**Linking a Server**

The following commands describe the steps required to link a user’s server code, `myserve.c` and a user’s application code, `myapp.c`, with the PV-WAVE archive library.

**NOTE** Compile and link options will vary by platform, and sometimes are site specific. Refer to the appropriate makefile in `$WAVE_DIR/util/rpc` for suggested compile and link options.

```bash
host> cc -c myserve.c
host> cc -c myapp.c
host> set arch='WAVE_DIR/bin/arch'
host> set ARCH=WAVE_DIR/bin/bin.$arch
host> cc -o myserve myserve.o myapp.o \
     $ARCH/rpc.$arch.a
```

With the following command, the server is started in the background on the desired server host machine.

```bash
host> myserve &
```

The user can now access the remote application by using the system function `CALL_UNIX` at the `WAVE>` prompt.

**Linking a Client**

In order to link a user application which is a client to communicate with PV-WAVE as a server, use the following commands:

```bash
host> cc -c myclient.c
host> cc -o myclient myclient.o \ 
     $ARCH/rpc.$arch.a
```

Once PV-WAVE has been started in the server state, the user can run his or her application client program.

```bash
host> myclient
```

**Using PV-WAVE as a Client: CALL_UNIX**

`CALL_UNIX` is designed to allow users to access applications on the same host machine running PV-WAVE or across a network to a remote host.

The usage for `CALL_UNIX` is:

```cpp
CALL_UNIX(param [, ...])
```
where `param` is a variable parameter of any PV-WAVE type.

For a complete description of CALL_UNIX, see the *PV-WAVE Reference*.

In most cases where PV-WAVE is the client, a user desires to send data from PV-WAVE to an existing application to perform specific tasks and return the result to PV-WAVE for further analysis. In any case, the user must write a server process which acts as an interface between PV-WAVE and his or her application. The following figure shows schematically how PV-WAVE is used as a client.

![Figure 2-2 Interapplication Communication: PV-WAVE as Client](image)

When using PV-WAVE as a client, users must first write a server to interface between PV-WAVE and their application. The server makes a call to the PV-WAVE RPC interface module `w_listen`, which is discussed in the section *Description of C Functions Used with CALL_UNIX* on page 63. When the server is started in
the background, \texttt{w\_listen} “sits and waits” for a procedure call message from PV-WAVE with the appropriate program number.

Once the server has been placed in this state, PV-WAVE as the client initiates the interapplication communication by calling CALL\_UNIX with the necessary parameters and keywords. CALL\_UNIX opens a socket for communication, packs the parameters into XDR format and then transfers the data to the server. The data is then “unpacked” by using \texttt{w\_get\_par}, which is discussed in the section \textit{Description of C Functions Used with CALL\_UNIX} on page 63.

It is essential that argument parameters passed to the server be in the proper order and be of the expected type and structure. The type, number and structure of parameters can be checked within a PV-WAVE procedure before the call to the server is made. The unpacked parameters can then be used in a call to the remote application.

After execution of the remote application is complete, the results are passed back to PV-WAVE using one of the following three routines:

- \texttt{w\_smpl\_reply} — Use this C routine when no passed parameters have been modified and there is only a single return value/array.
- \texttt{w\_send\_reply} — Use this C routine if the input parameters have been modified and must be sent back to PV-WAVE in addition to some return value/array.
- \texttt{w\_cmpnd\_reply} — Use this C routine to return multi-dimensional arrays.

Once the results have been returned to PV-WAVE, program control returns to PV-WAVE. The server can either exit or enter a new wait state.

\textbf{Description of C Functions Used with CALL\_UNIX}

As explained previously, the CALL\_UNIX function allows PV-WAVE to communicate with a user-written application. CALL\_UNIX sends parameters to a server process which then calls the user-written application.

The server uses the following C routines, which are discussed in detail later in this section:

- \texttt{w\_listen} — Connect with the process running PV-WAVE.
- \texttt{w\_get\_par} — Get the parameters.
- \texttt{w\_send\_reply, w\_smpl\_reply, or w\_cmpnd\_reply} — Send values and parameters back to PV-WAVE.

If an error occurs in a call to CALL\_UNIX, it returns \texttt{-1}. The function \texttt{ON\_IOERROR} can also be used to catch CALL\_UNIX errors.
**w_listen**

When placed in an external C source file, allows the routine to “sit and wait” until it is called.

**Usage**

```c
int w_listen (int program_number, char **user, char **procedure)
```

**Discussion**

When `w_listen` exits, the values of the parameters `user`, and `procedure` have been set. Use these strings in the external routine to control access to, and program flow in, the external routine.

**Returned Value**

`w_listen` returns a status of –1 if an error occurs, 1 otherwise.

**Input Parameters**

- `program_number` — A unique identifier, set within the external routine that enables `w_listen` to determine which calls from CALL_UNIX are intended for this particular server. If the value of `program_number` is the same as the value of `program_number` in another server, the communication with the first server will be lost. If communication with a server is lost, use the `kill -9` command.

- `user` — A pointer to a string pointer. The parameter `user` is set by the keyword `User` in the CALL_UNIX call and set by `w_listen`. The intended use of `user` is to control access to the external routine. The memory that holds the string has already been allocated.

- `procedure` — A pointer to a string pointer. The parameter `procedure` is set by the keyword `Procedure` in the CALL_UNIX call. The intended use of `procedure` is to control program flow in the external routine. The memory that holds the string has already been allocated.

**w_get_par**

Returns a specified parameter passed by CALL_UNIX.

**Usage**

```c
char *w_get_par (int param_number, type);
```
Parameters

\textit{param\_number} — The index of the list of parameters. \textit{param\_number} is zero-based, so a value of zero will cause \textit{w\_get\_par} to return a pointer to the first parameter, a value of one will cause \textit{w\_get\_par} to return a pointer to the second parameter and so on.

\textit{type} — The type of variable to be retrieved.

Discussion

Because of the way that functions are declared in C, \textit{w\_get\_par} returns a pointer to a char, which is then cast to a pointer to the type of parameter desired. The space pointed to by the return value is allocated and freed by the RPC software. When the external routine replies to the client, the allocated space is freed. Thus, if there is a need to save a value returned by \textit{w\_get\_par}, the value pointed to by the return value of \textit{w\_get\_par} must be copied into a variable. \textit{w\_get\_par} returns NULL if there is no such parameter or the specified parameter is not of the specified type.

The types are defined in the file \$\textsc{wave}\_\textsc{dir}/\textsc{util}/\textsc{rpc/}
\textsc{wave}\_\textsc{rpc\_extern.h}. For example, \texttt{TYP\_LONG} will cause \textit{w\_get\_par} to return a pointer to a long. To specify an array, perform a bitwise OR of the type of the array with the constant \texttt{TYP\_ARRAY}. For example, if the third parameter passed from \texttt{CALL\_UNIX} to the server is an array of long, you would use the following code to get the array:

\begin{verbatim}
long *example_long_array;
example_long_array = (long *) w_get_par (2, TYP_LONG | TYP_ARRAY);
\end{verbatim}

After the external routine has completed its processing, it will need to return information to the client. Three routines are supplied that are used to return the information. All three of these routines return a status value of –1 to indicate an error and 0 to indicate success.

\begin{quote}
\underline{NOTE} When one of these three routines is called, all the memory that the parameters occupied in the RPC software is freed. In the preceding example, the array that \texttt{example\_long\_array} points to will be freed.
\end{quote}

\texttt{w\_smpl\_reply} and \texttt{w\_send\_reply}

\begin{itemize}
\item \texttt{w\_smpl\_reply} — Sends a single value array as the return value of \texttt{CALL\_UNIX}.
\item \texttt{w\_send\_reply} — Is used when it is necessary to modify and return parameters that have been passed in from \texttt{CALL\_UNIX}.
\end{itemize}
Usage

```c
int w_simpl_reply (type, long number_of_items, char *items);
int w_send_reply (type, long number_of_items, char *items);
```

Parameters

type — The type of variable to be returned, as defined in the file:
$WAVE_DIR/util/rpc/wave_rpcExtern.h

To specify an array, perform a bitwise OR of the type of the array with the constant
TYP_ARRAY.

number_of_items — The number of items to be returned by the server. If an array
is being returned, number_of_items is the length of the array, otherwise,
number_of_items is set to one.

items — A pointer to the item(s) being returned. w_simpl_reply and
w_send_reply expect pointers to char, so they may have to be type cast.

Discussion

w_simpl_reply can be thought of as giving the CALL_UNIX function a call by
value parameter-passing mechanism. Only scalar values or single-dimension
arrays (vectors) or scalars can be returned by w_simpl_reply.

With w_send_reply, data passed to a server cannot “grow”. If the server was
passed FLTARR(100), the remote application cannot “grow” this to
FLTARR(1000) and return it. Thus, w_simpl_reply can be thought of as giv-
ing the CALL_UNIX function a call by reference parameter-passing mechanism.
Only single-dimension arrays or scalars can be returned by w_send_reply.

**w_cmpnd_reply**

Passes a multi-dimensional array back from the server.

Usage

```c
int w_cmpnd_reply (unsigned char reply_only, type, unsigned char
number_dimensions, long dimensions[ ], char *items);
```

Parameters

reply_only — A flag that determines if the parameters sent by the client are modi-
ﬁed and returned by the server. If reply_only has a value of one, then the parameters
sent from the client are not returned by the server (call by value). Any other value
of reply_only will cause the (possibly modified) parameters that were sent by the client to be sent by the server back to the client (call by reference).

type — The type of variable to be returned. The types are defined in the file $WAVE_DIR/util/rpc/wave_rpcExtern.h. To specify an array, perform a bitwise OR of the type of the array with the constant TYP_ARRAY.

number_dimensions — The size of the dimensions array. The maximum value is eight.

dimensions — An array containing the sizes of each of the array dimensions. To specify a 4 x 5 x 6 array, set number_dimensions to 3, and dimensions to [4,5,6].

items — A pointer to the item(s) being returned. w_cmpnd_reply expects pointers to char, so they may have to be type cast.

Example Server

An example server, test_server.c, is provided online in the directory WAVE_DIR/util/rpc. The file test_server.c contains three examples:

• **Example 1** — Accepts an array of long values. The array is passed to a function where each value is multiplied by two. The result is passed back and placed into the PV-WAVE variable newarray.

```
newarray = CALL_UNIX(proc='example_1', $1000L, lindgen(1000))
print, newarray
```

• **Example 2** — Accepts a string and returns a string. An array of strings is passed to the server and then printed. A new string is returned and placed in the PV-WAVE variable message.

```
strings=['one', 'two', 'three', 'four', 'five']
message=CALL_UNIX(proc='example_2', 5, strings)
print, message
```

• **Example 3** — Accepts a long array. This array is then filled by pseudo random numbers and returned. This example shows how to modify a passed parameter rather than always returning a result to a new variable. All parameters passed to the server in this example must be variables. Otherwise, when the server attempts to modify a parameter, PV-WAVE complains about not being able to store a value into a constant. old_array will contain an array of 1000 pseudo random numbers.

```
old_array=lonarr(1000)
old_arr_len=1000L
```
new_len=CALL_UNIX(old_arr_len, old_array, proc='example_3')
print, new_len, old_array

Example Using CALL_UNIX

See Example Procedure Using CALL_UNIX with PV-WAVE as Client on page 73 for example code showing a PV-WAVE client procedure that uses CALL_UNIX.

And see Example External C Routine as a Server on page 75 for an example of the corresponding server.

Using PV-WAVE as a Server: CALL_WAVE

CALL_WAVE is designed to allow users to access PV-WAVE from existing or planned applications on the same host machine running PV-WAVE or across a network to a remote host. In most cases where PV-WAVE is the server, the user desires to send data to PV-WAVE either to access PV-WAVE’s analytical routines or for plotting. In any case, the user must write a client process which acts as an interface between PV-WAVE and his or her application. See Figure 2-3 for an example of PV-WAVE as a server.

To run PV-WAVE as a server, start-up PV-WAVE and run a procedure which calls UNIX_LISTEN.

NOTE  UNIX_LISTEN and several other procedures discussed in this section are located in the WAVE_DIR/util/rpc directory.

In addition, the file $WAVE_DIR/util/rpc/RPC.README contains information on using RPCs and how PV-WAVE can be run as a server in the background, in another window, or on another host machine.

PV-WAVE is now in a “sit and wait” state. The user client application then initiates the interapplication communication by calling CALL_WAVE with the necessary parameters and keywords. CALL_WAVE opens a socket for communication, packs the arguments into XDR format and then transfers the data to PV-WAVE.

UNIX_LISTEN receives this procedure call message, unpacks the data from the client and then makes it available to PV-WAVE for some type of processing. UNIX_LISTEN can also return two strings to the PV-WAVE environment. These strings can contain specific PV-WAVE commands that specify the desired processing that should take place. Once this processing has been completed, PV-WAVE returns the results to the client application through UNIX_REPLY. When the
results have been returned, program control returns to the client while PV-WAVE can either exit or enter a new wait state.

**Figure 2-3** Interapplication Communication: PV-WAVE as Server
Description of C Functions Used with PV-WAVE as Server

CALL_WAVE

Calls a server and returns a pointer to a UT_VAR after the server has completed its task.

Usage

UT_VAR *CALL_WAVE(int argc, UT_VAR *argv[], char hostname[], int *unit,
int close_unit, int program, char *procedure, char *user, int time_out)

Parameters

(argc) — The number of values in argv. argc is also the number of parameters that
will be passed to PV-WAVE.

(argv) — An array of pointers to UT_VAR. The UT_VARs contain the parameters
that will be passed to PV-WAVE. It is up to the application to load these structures
properly.

(hostname) — The name of the machine that the server (PV-WAVE) is running on.
hostname is necessary because there could be several servers, each with the same
program number, but running on different machines.

(unit) — A number that maps to an RPC socket. The purpose of unit is to allow the
reuse of an open RPC socket, and thus avoid the overhead of re-opening the socket
each time CALL_WAVE is called. However, based on experience, this overhead is
not normally noticeable. If CALL_WAVE is called with NULL in the place of unit,
no value is returned and a new socket is opened and closed after CALL_WAVE
exits. If *unit is zero (i.e., unit == 0 and CALL_WAVE receives &unit),
then a socket is opened and the socket number is returned in *unit. If *unit is
greater than zero, socket number unit is used.

(close_unit) — A flag telling CALL_WAVE to close unit after CALL_WAVE
exits. If close_unit is non-zero, the unit is closed.

(program) — A number identifying which server the application is calling. program
is supplied in the C routine on the client. The value of program is the same value
of the keyword Program in the UNIX_LISTEN call. If the UNIX_LISTEN call did
not have the Program keyword in it, use zero for the value of program.

(procedure) — A string that is sent to the server. In the server, the UNIX_LISTEN
keyword Procedure is used to retrieve the string. user is intended to be used for
controlling program flow within the server.
user — A string that is sent to the server. In the server, the UNIX_LISTEN keyword
*User* is used to retrieve the string. *procedure* is intended to be used for controlling
access to the routines within the server.

time_out — The time, in seconds, that CALL_WAVE will wait for PV-WAVE to
complete its task. There is no value for infinite time.

**Discussion**

The function CALL_WAVE calls a server and returns a pointer to a UT_VAR after
the server has completed its task. UT_VAR is a structure for holding any valid type
of PV-WAVE variable. This UT_VAR pointer points to the value sent back in the
reply parameter of UNIX_REPLY. Upon receiving a pointer to a UT_VAR, the
application must determine the type of variable it references. The definitions of
UT_VAR and other types can be found in the file:

```
$WAVE_DIR/util/rpc/wave_rpc_extern.h
```
For convenience, here is the definition of UT_VAR:

```
typedef struct
{
    unsigned char type,
    unsigned char element_size;
    unsigned char n_dim,;
    long dim[MAX_ARRAY_DIM];
    UT_TYPES value;
} UT_VAR;
```

**PV-WAVE Functions Used with PV-WAVE as Server**

The UNIX_LISTEN and UNIX_REPLY functions are used when PV-WAVE is
acting as a server.

**UNIX_LISTEN**

The UNIX_LISTEN function allows PV-WAVE to be called by external routines
written in C. UNIX_LISTEN waits until an external routine calls the function
CALL_WAVE, and then returns the number of parameters with which it was called.

The parameters are accessed in the common block UT_COMMON, which is included
in the server routine with the command @UT_COMMON. The first parameter is
*ut_param0*, the second is *ut_param1*, and the thirtieth parameter is *ut_param29*. As
well as being returned, the number of parameters is also contained in the variable
*ut_num_params* in the UT_COMMON common block.
For more detailed information on UNIX_LISTEN and its keywords, see the PV-WAVE Reference.

**UNIX_REPLY**

When PV-WAVE has completed its processing and is ready to return information to the client, the function UNIX_REPLY is called. UNIX_REPLY has one parameter, \( reply \), which is a variable that is sent back to the client. The CALL_WAVE function returns a pointer to a UT_VAR variable with the value of \( reply \) to the client. A keyword, \( Return\_Params \), is used if the server must modify the incoming parameters and return them. The number of parameters that is sent back is the same as the number that came in. This number is tracked internally by PV-WAVE.

For more detailed information on UNIX_REPLY and its keywords, see the PV-WAVE Reference.

**Examples Using PV-WAVE as a Server**

See *Example Procedure Using UNIX_LISTEN and UNIX_REPLY with PV-WAVE as a Server* on page 84 for a code example that illustrates the interaction between a client application using CALL_WAVE and PV-WAVE as a server using the PV-WAVE functions UNIX_LISTEN and UNIX_REPLY.

An example client, *test_client.c*, is provided on the distribution tape in WAVE_DIR/util/rpc.

See also the code listings *Example C Routine with CALL_WAVE as a Client* on page 77 and *Example C Function to Load a UT_VAR* on page 81.

---

**Remote Procedure Call Examples**

This section contains the following RPC examples. Each example was referred to previously in this chapter:

- *Example Procedure Using CALL_UNIX with PV-WAVE as Client* on page 73.
- *Example External C Routine as a Server* on page 75.
- *Example C Routine with CALL_WAVE as a Client* on page 77.
- *Example C Function to Load a UT_VAR* on page 81.
- *Example Procedure Using UNIX_LISTEN and UNIX_REPLY with PV-WAVE as a Server* on page 84.
NOTE  If you are running on a Solaris system, note the following: When registering an RPC server on a program/version pair, Solaris checks that the user is allowed to register. If another user already registered a server for a program/version pair, Solaris will refuse the new registration. The owner of the pair must run rpcinfo with the -d option to delete the RPC service of the program/version pair before another user can register at that location. Use rpcinfo to check ownership.

Example Procedure Using CALL_UNIX with PV-WAVE as Client

You can find the following listed file in:

$WAVE_DIR/util/rpc/wave_client.pro

;This PV-WAVE procedure tests an array of samples from a single host or list of hosts. Each array will be a LONG array of length LENGTH. TIMES specifies the number of times to call each host.

PRO samples, length=length, times=times, $ hostlist=hostlist

IF (N_ELEMENTS(times) eq 0) THEN times=100
IF (N_ELEMENTS(length) eq 0 ) THEN length=1000
IF (N_ELEMENTS (hostlist) eq 0) THEN $
   hostlist= 'localhost'

;See if hostlist is a single host or a list of hosts
num_hosts = SIZE(hostlist)
IF(num_hosts(0) EQ 0) THEN BEGIN
   ;hostlist is a single hostname
   ;make it a string array of length 1
   name = hostlist
   hostlist = strarr(1)
   hostlist(0) = name
   num_hosts = 1
ENDIF ELSE BEGIN
   ;hostlist is an array of hosts
   ;num_hosts is now number of hosts
   num_hosts = num_hosts(1)
ENDIF

;connect is an array to store "unit" values
connect = LONARR(num_hosts)
;a LONG. It's important to pass a LONG, so
;that the SERVER will not "die", i.e.:
; Segmentation Violation : (core dumped)
length = LONG(length)
xpos = 0
ypos = 0
xsize = 300
ysize = 300

FOR i=0, num_hosts - 1 DO BEGIN
    hostname = hostlist(i)
    PRINT, 'Opening Window for ' + hostname
    WINDOW, i, colors=128, title=' Data from $ ' +hostname, xpos=xpos, ypos=ypos, $
    xsize=xsize, ysize=ysize
    xpos = xpos + xsize + 20
    IF (xpos GT 800) THEN BEGIN
        xpos = 0
        ypos = ypos + ysize + 30
    END
ENDFOR

IF(!d.n_colors GT 2) THEN BEGIN
    !p.background = 12
    !p.color = 127
    LOADCT,4
ENDIF

FOR j=1, times - 1 DO BEGIN
    ;Call each host and save the connection "unit"
    FOR i=0, num_hosts - 1 DO BEGIN
        ; plot in the correct window
        WSET, i
    END

    ;A temporary variable is needed because
;PV-WAVE is not able to store into an
;expression
unit = connect (i)

;This is the call to the external
;routine/server
array = CALL_UNIX (length, hostname= $ hostlist (i), unit=unit)

;After the first call, connect (i) stays
;the same
connect (i) = unit
PLOT, array

;Empty the graphics buffer
EMPTY
ENDFOR
ENDFOR

;This is the last time; close all units
FOR i=0, num_hosts - 1 DO BEGIN
    WSET, i
    unit = connect (i)

    ;connect(i) is closed after this call

    array = CALL_UNIX (length, hostname= $ hostlist (i), unit=unit, /close)
    PLOT, array
    EMPTY
ENDFOR
END

**Example External C Routine as a Server**

You can find the following listed file in:
$WAVE_DIR/util/rpc/samples.c

# include "wave_rpcExtern.h"

/****************************

This is an example of a simple server. It does little error checking and thus can core dump. If, however, PV-WAVE always sends the correct parameters, there is no problem.
This program is intended to respond to the PV-WAVE statement:

ARRAY = CALL_UNIX(num_samples)

where num_samples is a LONG. The PV-WAVE procedure SAMPLES.PRO gives an example of how to access this server.

#define MAX_SAMPLES 100000
main (argc, argv)
int argc;
char *argv[];
{
    int i, id;
    char *user, *proc;
    long num_samples, samples[MAX_SAMPLES];
    /* If no program number is specified, 0 is the default */
    if (argv[1] != NULL) {
        id = atoi (argv[1]);
    } else {
        id = 0;
    }
    /* Forever or until "kill -9" */
    while (1) {
        /* Listen for PV-WAVE to call via
         "CALL_UNIX(params)" */
        w_listen (id, &user, &proc);
        /* WAVE_LONG is a macro that evaluates to:
         *(long *)w_get_par(0, TYP_LONG);
         w_get_param will return NULL if the parameter is not a long. The
         NULL pointer could cause a segmentation violation and cause a
core dump. It is ok, however, as long as PV-WAVE passes expected
         values. */
        num_samples = WAVE_LONG;
        /* Generate the samples. rand() returns a random number */
        for (i = 0; i < num_samples; i++)
            samples[i] = rand();
        /* Reply to PV-WAVE with the array "samples". Only "num_samples" are
         used. PV-WAVE will receive a LONG array num_samples in length.
         W_SMPL_REPLY() does not return the passed parameters to PV-WAVE.*/
Since this routine didn’t modify any of the parameters there is no need to return them */

w_smpl_reply (TYP_LONG, num_samples, samples);
}
}

**Example C Routine with CALL_WAVE as a Client**

You can find the following listed file in:

$WAVE_DIR/util/rpc/test_client.c

`#include "wave_rpcExtern.h"
/*****************************/

This is an example of a client C program that calls a PV-WAVE server. First, a LONG array is generated with "rand ()". The array is then sent to PV-WAVE along with a smoothing factor and a plot title. The following PV-WAVE procedure is used to accept the information, smooth the array, and return it as the result of "CALL_WAVE ()". Optionally, PV-WAVE will plot the smoothed array.

This program can be started in two fashions: (program is the name of the executable version of this program)

```
program
or
program program_number hostname
```

The program number is needed if the PV-WAVE SERVER was started with a program number other than the default value of 0. The hostname is needed if the server is not running on "localhost".

`/*****************************/
/* Length of the array to be smoothed */
#define ARRAY_LENGTH 200
/* Window to use for smoothing */
#define SMOOTH_FACTOR 5
main (argc, argv)
int argc;
char *argv[];
{
    int make_single_array ();
    /* Code to create a UT_VAR; follows main() */
int i, id, j, unit, status, times;
char *title;
UT_VAR *retval;
/* Variables are passed to WAVE using "UT_VAR"s; see
wave_rpcExtern.h for definition. While only 3 UT_VARs are used
in this example, up to 30 variables may be passed. ut_arr[3] and
ut_arr[4] are not used in this example and are not necessary and
are there for convenience. */
static UT_VAR ut_arr[5];
/* "call_wave" expects an array of UT_VAR pointers */
static UT_VAR *argptr[] = {
&ut_arr[0],
&ut_arr[1],
&ut_arr[2],
&ut_arr[3],
&ut_arr[4]};
/* While the actual user of "procedure" and "user" are actually
determined by the SERVER, it is suggested that the "procedure"
parameter be used to select a particular function inside the
SERVER and the "user" parameter be used as site-specific secu-
rity. */
char *user, *procedure, *hostname;
long num_neighbors, long_arr[ARRAY_LENGTH];
/* If no pargram number was specified, use 0 */
if (argc >= 2) {
    id = atoi (argv[1]);
} else {
    id = 0;
}
/* If no hostname was specified, use "localhost" */
if (argc >= 3) {
    hostname = argv[2];
} else {
    hostname = "localhost";
}
/* Fake user data; generally used for site-specific security */
user = "Test Client User Data";
/* Function inside the server that this CLIENT wishes to access */
procedure = "SMOOTH";
/* Title for plot */
title = "Smoothed Random Numbers";
/* Window to use for smoothing */
num_neighbors = SMOOTH_FACTOR;
/* Figure how many times to call the server */
printf("Number of times to call SERVER %s :", hostname);
scanf("%d", &times);
for (j = 0; j < times; j++) {
  /* Generate a random array */
  for (i = 0; i < ARRAY_LENGTH; i++) {
    long_arr[i] = rand();
  }
  /* Create a UT_VAR with smoothing factor */
  status = make_single_array (argptr[0],
    TYP_LONG, 1, &num_neighbors);
  if (status < 0) {
    sprintf(stderr,"Error building UT_VAR\n");
    exit (1);
  }
  /* Create UT_VAR with long array */
  status = make_single_array (argptr[1],
    TYP_LONG | TYP_ARRAY, ARRAY_LENGTH,
    long_arr);
  if (status < 0) {
    sprintf (stderr, "Error building UT_VAR\n");
    exit (1);
  }
  /* Make UT_VAR with title */
  status = make_single_array (argptr[2],
    TYP_STRING, 1, &title);
  if (status < 0) {
    sprintf(stderr,"Error building UT_VAR\n");
    exit (1);
  }
This is the call to the SERVER. The SERVER’s response will be pointed to by "retval".

CALL_WAVE() is used in by :
RETURN_VALUE = CALL_WAVE(number of UT_VARs in parameter array, parameter array (UT_VAR *array[]), name of host to call, unit (NULL means don’t care) close (if 1, close specified unit), procedure name, user data, timeout in secs (0 means use default))

The unit parameter provides a way to maintain a connection between the SERVER and CLIENT. If unit is non-null and points to an integer = 0, then once the connection is established it is left open. Further calls using that unit will use the existing connection instead of creating a new one. If close is 1, then the connection is closed after the call.

Using the unit parameter causes extra file descriptors to be left open inside the UNDERTOE CLIENT. Since there is a limit on the number of open file descriptors, don’t leave too many units open. Using the unit parameter is useful when establishing the connection takes a long time. Tests on several machines have shown that this time is generally much less than a second. That delay is only a significant portion of the total transfer time if the total data transfer is < 10,000 bytes and the SERVER’s computation time is negligible. So to avoid some added housekeeping, use NULL

retval = (UT_VAR *) call_wave (3, argptr, hostname, NULL, 0, id, procedure, user, 0);
/* Check the type of the returned parameter */
/* If it is not an error and it’s an array, print it */
if (retval->type == (TYP_LONG | TYP_ARRAY))
{
    for (i = 0; i < ARRAY_LENGTH; i++) {
        /* value.array is defined as (char *) so cast to the appropriate type. */
        printf("%d ", ((long *)
            retval->value.array)[i]);
    }
}
else {
    printf("SERVER did not return \n");
    printf("expected value\n");
    printf ("Returned from call_wave \n");
}
printf("retval type is \%d \n", 
    retval->type);
}
}
}

**Example C Function to Load a UT_VAR**

You can find the following listed file in:

$WAVE_DIR/util/rpc/test_client.c

```c
int make_single_array (ut_ptr, type, length, array_ptr)

unsigned char type; /* type of UT_VAR */
long length; /* length of array */
char *array_ptr; /* pointer to data */
UT_VAR *ut_ptr; /* pointer to UT_VAR */
{
    /* Sample routine to build UT_VARs. This routine will only build 
     * UT_VARs that are single dimensioned. To build a multi-dimensional 
     * array set n_dim and dim[], i.e.:
     * A three dimensional array that is 4 by 5 by 6 would be:
     *     ut_ptr->n_dim = 3;
     *     ut_ptr->dim[0] = 4;
     *     ut_ptr->dim[1] = 5;
     *     ut_ptr->dim[2] = 6;
     */
    /* switch on type ignoring the TYP_ARRAY 
     * bit */
    switch (type & SIMPLE_MASK) {
    case TYP_BYTE:
        ut_ptr->type = TYP_BYTE;
        ut_ptr->element_size = sizeof (char);
        ut_ptr->n_dim = 1;
        ut_ptr->dim[0] = length;
        if (type & TYP_ARRAY) {
            ut_ptr->type = ut_ptr->type | TYP_ARRAY;
            ut_ptr->value.array = array_ptr;
        }
        else {
```
/* array_ptr is a (char *), cast to proper type */
    ut_ptr->value.c = *(unsigned char *)
        array_ptr;
}
break;

case TYP_INT:
    ut_ptr->type = TYP_INT;
    ut_ptr->element_size = sizeof (short);
    ut_ptr->n_dim = 1;
    ut_ptr->dim[0] = length;
    if (type & TYP_ARRAY) {
        ut_ptr->type = ut_ptr->type | TYP_ARRAY;
        ut_ptr->value.array = array_ptr;
    } else {
        /* array_ptr is a (char *), cast to proper type */
        ut_ptr->value.i = *(short *) array_ptr;
    }
break;

case TYP_LONG:
    ut_ptr->type = TYP_LONG;
    ut_ptr->element_size = sizeof (long);
    ut_ptr->n_dim = 1;
    ut_ptr->dim[0] = length;
    if (type & TYP_ARRAY) {
        ut_ptr->type = ut_ptr->type | TYP_ARRAY;
        ut_ptr->value.array = array_ptr;
    } else {
        /* array_ptr is a (char *), cast to proper type */
        ut_ptr->value.l = *(long *) array_ptr;
    }
break;

case TYP_DOUBLE:
    ut_ptr->type = TYP_DOUBLE;
    ut_ptr->element_size = sizeof (double);
    ut_ptr->n_dim = 1;
    ut_ptr->dim[0] = length;
if (type & TYP_ARRAY) {
    ut_ptr->type = ut_ptr->type | TYP_ARRAY;
    ut_ptr->value.array = array_ptr;
}
else {
    /* array_ptr is a (char *), cast to proper type */
    ut_ptr->value.d = *(double *) array_ptr;
}
break;

case TYP_FLOAT:
    ut_ptr->type = TYP_FLOAT;
    ut_ptr->element_size = sizeof (float);
    ut_ptr->n_dim = 1;
    ut_ptr->dim[0] = length;
    if (type & TYP_ARRAY) {
        ut_ptr->type = ut_ptr->type | TYP_ARRAY;
        ut_ptr->value.array = array_ptr;
    }
    else {
        /* array_ptr is a (char *), cast to proper type */
        ut_ptr->value.f = *(float *) array_ptr;
    }
    break;

case TYP_COMPLEX:
    ut_ptr->type = TYP_COMPLEX;
    ut_ptr->element_size = sizeof (COMPLEX);
    ut_ptr->n_dim = 1;
    ut_ptr->dim[0] = length;
    if (type & TYP_ARRAY) {
        ut_ptr->type = ut_ptr->type | TYP_ARRAY;
        ut_ptr->value.array = array_ptr;
    }
    else {
        /* see wave_rpcExtern.h for definition of COMPLEX array_ptr is a
           * (char *), cast to proper type */
        ut_ptr->value.cmp.r = ((COMPLEX *)
            array_ptr)->r;
ut_ptr->value.cmp.i = ((COMPLEX *)
array_ptr)->i;
}
break;
case TYP_STRING:
    ut_ptr->type = TYP_STRING;
    ut_ptr->element_size = sizeof(char);
    ut_ptr->n_dim = 1;
    ut_ptr->dim[0] = length;
    if (type & TYP_ARRAY) {
        /* a string array is an array of (char *) */
        ut_ptr->element_size = sizeof(char*);
        ut_ptr->type = ut_ptr->type | TYP_ARRAY;
        ut_ptr->value.array = array_ptr;
    }
    else {
        /* array_ptr is a (char *), cast to proper type */
        ut_ptr->value.string = *(char **)array_ptr;
    }
break;
default:
    /* unsupported types: at present
TYP_STRUCTURE is not supported */
return (-1);
break;
}

Example Procedure Using UNIX_LISTEN and UNIX_REPLY with PV-WAVE as a Server

You can find the following listed file in:
$WAVE_DIR/util/rpc/example_server.pro

;This is an example of using PV-WAVE as the
;SERVER. Times is the number of times to loop
;through the SERVER. Show specifies that the
;resulting array should be plotted.
PRO example_server, Times=times, Show=show
@UT_COMMON
prog = 0
;Default is to loop through only once
IF (n_elements(times) EQ 0) then times = 1
WHILE(1) DO BEGIN
FOR i = 1, times DO BEGIN
;Listen for a call using the default program
;number zero. Return the received "procedure"
;string into the variable "proc"
  n = UNIX_LISTEN(program = prog, procedure = $ proc)
;Make sure the proc is SMOOTH and we have the
;correct number of parameters
  IF (n EQ 3) and (proc EQ 'SMOOTH') THEN BEGIN
    ;Smooth the array
    array = SMOOTH(ut_param1, ut_param0)
    ;Return the array to the caller but do not
    ;return the parameters. To return the
    ;parameters, the call would be:
    
    status = UNIX_REPLY(array)
  status = UNIX_REPLY(array)
;Only plot when told to do so
  IF (keyword_set(show)) THEN BEGIN
    !P.title = ut_param2
    PLOT, array
    EMPTY
  ENDFILE
ENDIF ELSE BEGIN
;The server did not get the correct proc or
;did not get the correct number of parameters
PRINT, 'EXAMPLE_SERVER MISMATCH: ' + $
  'PROC = ', proc, ' N_PARAMS = ', n
  status = UNIX_REPLY(-1)
ENDElse
ENDFOR
ENDWHILE
END
**Interapplication Communication Using the Socket OPI**

The PV-WAVE Socket OPI greatly simplifies the programming required to write PV-WAVE client and server programs that communicate through sockets. Using the Socket OPI routines, you can write client and server applications entirely in PV-WAVE.

Sockets allow programmers to treat network connections as a stream of bytes that can be read from or written to. Sockets handle such low-level tasks of data transmission and network addresses, thereby freeing network programmers to concentrate primarily on network applications.

The PV-WAVE Socket routines allow PV-WAVE applications to:
- listen for connections (server)
- initiate connections (client)
- read and write data through connections (client and server)
- terminate connections (client and server)

For detailed information on the Socket OPI routines, see Chapter 2, *New Commands*.

The following table lists the Socket OPI routines and indicates whether a specific routine is typically used in client applications, server applications, or both.

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Purpose</th>
<th>Where Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCKET_ACCEPT Function</td>
<td>Waits for a connection on a socket to occur.</td>
<td>Server</td>
</tr>
<tr>
<td>SOCKET_CLOSE Procedure</td>
<td>Closes a socket connection.</td>
<td>Both</td>
</tr>
<tr>
<td>SOCKET_CONNECT Function</td>
<td>Connects to a socket at a given host and port.</td>
<td>Client</td>
</tr>
<tr>
<td>SOCKET_GETPORT Function</td>
<td>Returns the socket port number for the specified socket.</td>
<td>Server</td>
</tr>
<tr>
<td>SOCKET_INIT Function</td>
<td>Initializes a socket connection.</td>
<td>Server</td>
</tr>
<tr>
<td>SOCKET_READ Function</td>
<td>Reads data from a socket connection.</td>
<td>Both</td>
</tr>
<tr>
<td>SOCKET_WRITE Procedure</td>
<td>Writes data to a socket connection.</td>
<td>Both</td>
</tr>
</tbody>
</table>
Overview of a Client-Server Model Using Sockets

The following figure shows a typical client-server model using the PV-WAVE Socket OPI.

On the server, socket port 1800 is initialized with the SOCKET_INIT function. The SOCKET_ACCEPT function waits for a connection. The client initializes a connection with the SOCKET_CONNECTION function. This function specifies a host address (for example, www.vni.com) and a port number.

The SOCKET_READ and SOCKET_WRITE routines allow data to be sent between the client and server, through the socket connection. SOCKET_WRITE sends a byte array through an established socket connection. SOCKET_READ retrieves the byte array.

Loading the Socket OPI

Before you can use the PV-WAVE socket routines, the Socket OPI (Optional Programming Interface) must be loaded on the server and the client.

To load the Socket OPI execute the program file (batch file) SOCKET_STARTUP at the PV-WAVE command line. For example:

WAVE> @SOCKET_STARTUP

Initializing a Socket

The SOCKET_INIT function allows you to specify which socket port on the server machine will be used to “listen” for incoming client connections. This function
returns a handle for the socket. This handle is used by other Socket OPI functions to refer to the socket.

In the following lines, socket port 1800 is initialized with SOCKET_INIT, and then the resulting socket descriptor is passed to the SOCKET_ACCEPT function, which listens for client connections.

\[
\text{socketID}=\text{SOCKET\_INIT}(1800) \\
\text{connection}=\text{SOCKET\_ACCEPT(socketID)}
\]

### Listening for a Socket Connection

After a socket is initialized, call the SOCKET_ACCEPT function to wait for client connections. SOCKET_ACCEPT blocks further execution of the current application until a connection is received. When a connection is received, the SOCKET_ACCEPT function returns a handle for the connection. This handle is then used to identify the connection in other socket routines, like SOCKET_READ and SOCKET_WRITE.

In the following lines, SOCKET_READ is used to read data from a socket connection on the server.

\[
\text{socketID}=\text{SOCKET\_INIT}(1800) \\
\text{myconnection}=\text{SOCKET\_ACCEPT(socketID)} \\
\text{mydata}=\text{BYTEARR}(100) \\
\text{nbytes}=\text{SOCKET\_READ(myconnection, mydata)}
\]

### Connecting to a Socket

Client applications use the SOCKET_CONNECT function to connect to a server that is listening for a connection (that is, SOCKET_ACCEPT must be running on the server). SOCKET_CONNECT specifies a host address (host name or IP address) and a port number. This function returns a connection handle, which is used in SOCKET_READ and SOCKET_WRITE to identify the connection.

The following call to SOCKET_CONNECT attempts to make a connection to port 80 on the Visual Numerics Web server. Then, SOCKET_WRITE sends data to the remote server using the established connection.

\[
\text{myconnection}=\text{SOCKET\_CONNECT(‘www.vni.com’, 80)} \\
\text{mydata}=\text{BYTE(‘GET / HTTP/1.0\015\012\015\012‘)} \\
\text{SOCKET\_WRITE, myconnection, mydata}
\]
Reading and Writing Data Between Client and Server

After a socket connection is established between a client and server, the routines SOCKET_WRITE and SOCKET_READ are the mechanisms by which data is sent and retrieved.

SOCKET_WRITE is a procedure that sends a byte array through the socket connection. SOCKET_READ retrieves a byte array on the other side of the connection.

It is up to the client and server software developers to decide how to encode and decode the data that is sent and received by SOCKET_WRITE and SOCKET_READ. The only requirement is that the data be a byte array.

TIP Use data extraction functions (BYTE, FLOAT, FIX, and so on) and BYTEORDER with the Htons and Htonl keywords to format data for network transmission.

Closing a Socket Connection

To close a socket connection, call SOCKET_CLOSE. This procedure takes one argument, the connection handle (the result returned from SOCKET_CONNECT).

```
myconnection=SOCKET_CONNECT(‘www.vni.com’, 80)
SOCKET_WRITE, myconnection, mydata
SOCKET_CLOSE, myconnection
```

Example

This example demonstrates a simple series of transactions between a client and a server program. The client sends a string to the server, and the server prints the string. The server then returns a string, which the client prints. A FOR loop repeats this sending and receiving pattern a total of three times.

NOTE All data transmitted through PV-WAVE socket connections must be in the form of a byte array.

The Server

This server program reads data sent from the client, prints the data, and sends data back to the client.
PRO SERVER
    port = 1500
    socket = SOCKET_INIT(port)
    connection = SOCKET_ACCEPT(socket)
    FOR i = 0,2 DO BEGIN
        data = BYTARR(15)
        nbytes = SOCKET_READ(connection,data)
        PRINT, 'SERVER received: ', STRING(data)
        data = BYTE('Server String ' + STRTRIM(STRING(i),2))
        SOCKET_WRITE, connection, data
        PRINT, 'SERVER sent: ', data
    ENDFOR
    SOCKET_CLOSE, connection
    SOCKET_CLOSE, socket
END

The Client Program

This client program sends data to the server, then reads and prints data returned from the server.

PRO CLIENT
    host = 'localhost'
    port = 1500
    socket = SOCKET_CONNECT(host,port)
    IF socket EQ -1 OR socket EQ -2 THEN BEGIN
        PRINT, 'SOCKET_CONNECT failed with return code: ', socket
        RETURN
    ENDFI
    FOR i = 0,2 DO BEGIN
        data = BYTE('Client String ' + STRTRIM(STRING(i),2))
        PRINT, 'CLIENT sending: ', data
        SOCKET_WRITE, socket, data
        data = BYTARR(15)
        nbytes = SOCKET_READ(socket,data)
        PRINT, 'CLIENT received: ', STRING(data)
WAIT, 1
ENDFOR

SOCKET_CLOSE, socket
END

Running the Example

To run this example, do the following:

Step 1  Copy the server program into a file called server.pro.

Step 2  Copy the client program into a file called client.pro.

Step 3  Start PV-WAVE and enter the following command:
        WAVE> .RUN server.pro

Step 4  Start another PV-WAVE session and enter the following command:
        WAVE> .RUN client.pro

Step 5  Start the server program in the server session window by typing server
        at the WAVE> prompt.

Step 6  Start the client program in the client session window by typing client
        at the WAVE> prompt.

Client Program Output

In the client window, the following output appears:

CLIENT sending:  67 108 105 101 110 116 32 83 116 114 105 110 103 32 48
CLIENT received: Server String 0

CLIENT sending:  67 108 105 101 110 116 32 83 116 114 105 110 103 32 49
CLIENT received: Server String 1

CLIENT sending:  67 108 105 101 110 116 32 83 116 114 105 110 103 32 50
CLIENT received: Server String 2

Server Program Output

In the server window, the following output appears:

SERVER received: Client String 0

SERVER sent:  83 101 114 118 101 114 32 83 116 114 105 110 103 32 48
Writing a Continuously Running Server

In the previous example, the server program stops running after the client closes its socket connection. In some cases, you may want the server to continue running and to wait for new client connections.

The key to writing a continuously running server is to enclose the SOCKET_ACCEPT function and subsequent socket functions in an infinite loop.

In the following example, the previously discussed program server.pro is modified so that it does not exit after the client connection is closed. Instead, the program loops back to the SOCKET_ACCEPT statement and waits for another client connection. Changes to the original program (only three lines) are shown in boldface type.

```
PRO SERVER
  port = 1500
  socket = SOCKET_INIT(port)
  connection = SOCKET_ACCEPT(socket)
  j=1
  WHILE j EQ 1 DO BEGIN
    FOR i = 0,2 DO BEGIN
      data = BYTARR(15)
      nbytes = SOCKET_READ(connection,data)
      PRINT, 'SERVER received: ', STRING(data)
      data = BYTE('Server String ' + STRTRIM(STRING(i),2))
      SOCKET_WRITE, connection, data
      PRINT, 'SERVER sent: ', data
    ENDFOR
    SOCKET_CLOSE, connection
  ENDFILE
  SOCKET_CLOSE, socket
END
```
Interapplication Communication for Windows

PV-WAVE provides a variety of methods for interapplication communication under Windows. For example:

- PV-WAVE can execute external programs and exchange data with them.
- External programs can call PV-WAVE to perform graphics, data manipulation, and other functions. Depending on the method used, the communication can be unidirectional or bidirectional.

Methods of Interapplication Communication

The following table summarizes the communication methods that can be used between PV-WAVE and other external applications. This table and the following section, Choosing the Best Method on page 94, can help you to determine the most appropriate method of interapplication communication to accomplish a desired task. Each method listed is described in detail later in this chapter.

<table>
<thead>
<tr>
<th>Method</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINKNLOAD</td>
<td>A system routine that allows PV-WAVE to call an external function via dynamic linked libraries. It is the simplest method for calling your own C code from PV-WAVE. Allows the transfer of binary data. See Using LINKNLOAD to Call External Programs on page 96.</td>
</tr>
</tbody>
</table>
Choosing the Best Method

It is important to select the most appropriate method of interapplication communication for your particular needs. Choosing the wrong method often requires much more work than is necessary to accomplish a given task.

This section describes typical scenarios where some kind of interapplication communication is required. After each scenario is described (in italics), a suitable solution for interapplication communication is suggested.

*I’m running PV-WAVE, and I want to execute an external program I’ve written. I’m not really concerned about returning anything to PV-WAVE.*

This is the simplest case of interapplication communication. The SPAWN procedure is the best choice. SPAWN executes an external program, or an operating system command, from PV-WAVE. SPAWN is described in the previous chapter.

<table>
<thead>
<tr>
<th>Method</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>cwavec</td>
<td>A routine that allows a dynamically linked C program to access PV-WAVE. Data is transferred between the C program and PV-WAVE via the wavevars routine (or PV-WAVE variable handles if you use the Option Programming Interface).</td>
</tr>
<tr>
<td>cwavefor</td>
<td>A routine that allows a dynamically linked FORTRAN program to access PV-WAVE. Data is transferred between the FORTRAN program and PV-WAVE via the wavevars routine</td>
</tr>
<tr>
<td>DDE</td>
<td>Dynamic Data Exchange functions allow client applications to call PV-WAVE and modify variables, query variables, and execute functions and procedures.</td>
</tr>
<tr>
<td>Option Programming Interface (OPI)</td>
<td>For developers who want to create optional modules that can be loaded explicitly by any PV-WAVE user. These optional modules can be written in C or FORTRAN, and can contain new system functions or other primitives. For information on OPI, see the PV-WAVE Programmer’s Guide.</td>
</tr>
<tr>
<td>Socket OPI</td>
<td>Allows you to treat network connections as streams of bytes that can be read from or written to. With the Socket OPI, you can write client and server applications entirely in PV-WAVE. The Socket OPI is described in the previous chapter; see Interapplication Communication Using the Socket OPI on page 86.</td>
</tr>
</tbody>
</table>
I want to establish a connection between PV-WAVE and a client application. I want the client to be able to send data to PV-WAVE and execute PV-WAVE functions, such as its high level graphics display functions. I don’t want to link PV-WAVE with the client application.

Dynamic Data Exchange (DDE) allows you to register PV-WAVE as a server application with any client application. The DDE protocol permits communication between applications without actually linking the applications into a single executable application. Once PV-WAVE is registered as a server, the client can exchange data with PV-WAVE and execute PV-WAVE functions.

I wrote a C program, and I want to be able to link it dynamically with PV-WAVE. My program needs to be able to access data directly from the data space of PV-WAVE. When my program is finished running, I want control returned back to PV-WAVE.

The LINKNLOAD procedure is the simplest method for attaching your own code to PV-WAVE. LINKNLOAD is a PV-WAVE system procedure that calls a function in a Dynamic Link Library (DLL). When used in conjunction with the wavevars function, data can be passed back and forth between the user-written routine and PV-WAVE.

For information on LINKNLOAD, see Using LINKNLOAD to Call External Programs on page 96. For information on the data transfer function wavevars see Accessing Data in PV-WAVE Variables on page 109.

NOTE See also the section Using the Option Programming Interface on page 113. The Option Programming Interface (OPI) functions allow user-written C code to access PV-WAVE variables and use other PV-WAVE functionality. OPI provides greater flexibility and control than wavevars.

I want to be able to call PV-WAVE from a C or FORTRAN program I’ve written. I want the program to be dynamically linked with PV-WAVE.

The cwavec function allows a dynamically linked C program to access PV-WAVE’s data space. Data is transferred between the C program and PV-WAVE via the wavevars routine. In addition, the cwavefor function allows a dynamically linked FORTRAN program to access PV-WAVE’s data space.

For information on cwavec see Calling PV-WAVE as a Dynamically Linked Program on page 98. For information on the data transfer function wavevars see Accessing Data in PV-WAVE Variables on page 109. The cwavefor routine is discussed in the section cwavefor: Calling PV-WAVE from a FORTRAN Program on page 103.
NOTE See also the section Using the Option Programming Interface on page 113. The Option Programming Interface (OPI) functions allow user-written C code to access PV-WAVE variables and use other PV-WAVE functionality. OPI provides greater flexibility and control than wavevars.

Using LINKNLOAD to Call External Programs

The LINKNLOAD function provides simplified access to external routines in Dynamic Link Libraries (DLLs). LINKNLOAD calls a function in a DLL and returns a scalar value. Parameters are passed through PV-WAVE to the specified external function by reference, thus allowing the external function to alter values of PV-WAVE variables. It is the simplest method for attaching your own C code to PV-WAVE.

Usage

\[
\text{result} = \text{LINKNLOAD}(\text{object}, \text{symbol} [, \text{param}_1, ..., \text{param}_n])
\]

Parameters

- **object** — A string specifying the filename, optionally including file path, of the DLL to be linked and loaded.
- **symbol** — A string specifying the function name (symbol entry point) to be invoked in the DLL file.
- **param\_i** — The data to be passed as a parameter to the function.

For more detailed information on the LINKNLOAD parameters and optional keywords see the discussion of LINKNLOAD in the PV-WAVE Reference.

Discussion

LINKNLOAD lets you call a C function from PV-WAVE almost as if you were calling a PV-WAVE function. The called function can obtain information from PV-WAVE through passed parameters or by accessing PV-WAVE’s variables directly (see Accessing Data in PV-WAVE Variables on page 109).

Any PV-WAVE data type, except a structure, can be passed as a parameter to a C routine. Parameters are always passed by reference (not by value), and thus it is up to the programmer’s discretion whether or not the C function alters the parameter’s value. Parameters are passed in the traditional C fashion of \text{argc} and \text{argv}. The C
function must know the type to expect for each parameter and must cast it to a C variable of the correct type.

**NOTE** Make sure the number, type, and dimension of the parameters passed to the external function match what the external function expects (this can most easily be done from within PV-WAVE before calling LINKNLOAD). Furthermore, the length of string parameters must not be altered and multi-dimensional arrays are flattened to one-dimensional arrays.

### Accessing the Data in PV-WAVE Variables

The `wavevars` function can be used to access the results generated by PV-WAVE in a user-written application called with LINKNLOAD. `wavevars` is a C function that can be invoked from code that is dynamically linked to PV-WAVE to obtain data from PV-WAVE’s data variable space.

The `wavevars` calling sequence is:

```c
result = wavevars(&argc, &argv, &argp);
```

For detailed information on `wavevars`, see *Accessing Data in PV-WAVE Variables* on page 109.

**NOTE** See also the section *Using the Option Programming Interface* on page 113. The Option Programming Interface (OPI) functions allow user-written C code to access PV-WAVE variables and use other PV-WAVE functionality. OPI provides greater flexibility and control than `wavevars`.

### Example 1: Calling a C Program

The C code referred to in this example can be found online. You can print the program or view it online using any text editor. The example program is in the file:

```
%WAVE_DIR%\demo\interapp\win32\linknload\example.c
```

In this example, parameters are passed using the conventional `argc`, `argv` strategy. `argc` indicates the number of data pointers which are passed from PV-WAVE within the array of pointers called `argv`. The pointers in `argv` can be cast to the desired type as the example program demonstrates.
Building the DLL File

The makefile (called makefile) creates the DLL file used by LINKNLOAD to link the C function to PV-WAVE at runtime. This makefile is in the same directory as the source file example.c. At an MS-DOS Command window prompt, enter the following command to build the DLL:

```
nmake
```

**NOTE** You must have a supported Windows C compiler on your system for this makefile to execute properly.

Accessing the External Function with LINKNLOAD

The following PV-WAVE code demonstrates how the C function defined in this example could be invoked.

```
ln = LINKNLOAD('example.dll','WaveParams', byte(1),2,long(3), float(4),double(5), complex(6,7), 'eight')
```

The resulting output is:

```
1 2 3 4.000000 5.000000 <6.000000,7.000000i> 'eight'
```

Using the INFO command, you can see that LINKNLOAD returns the scalar value 1.

```
INFO, ln
LN LONG = 1
```

The example program works with both scalars and arrays since the actual C program above only looks at the first element in the array and since PV-WAVE collapses multi-dimensional arrays to one-dimensional arrays:

```
ln = LINKNLOAD('example.dll','WaveParams', [byte(1)], [[2,3],[4,5]], [long(3)], [float(4)],[double(5)],[complex(6,7)], ['eight'])
```

The resulting output is:

```
1 2 3 4.000000 5.000000 <6.000000,7.000000i> 'eight'
```

Calling PV-WAVE as a Dynamically Linked Program

An application written in C or FORTRAN can be linked with the PV-WAVE Dynamic Link Libraries (DLLs), creating an application that can execute PV-WAVE commands at runtime. The C or FORTRAN application passes PV-WAVE commands to the entry points cwavec or cwavefor in the PV-WAVE DLL.
cwavec: Calling PV-WAVE from a C Program

The routine cwavec, discussed in detail in this section, is the C application entry point to a PV-WAVE DLL.

Usage

\[
\text{istat} = \text{cwavec}(\text{action, numcmds, cmds})
\]

Parameters

\text{action} — Specifies how you wish PV-WAVE to execute. It can have one of the following values:

- \text{action}=1 — Run normally. You are interactively prompted for input and execution continues until you enter the end-of-file character or issue the EXIT command. At this point, cwavec returns with a value of 1. Once cwavec has been called in this mode, it should not be called again.
- \text{action}=2 — Execute the commands supplied by \text{cmds} array and return. The return value is the value of the !Error system variable. The cwavec routine can be called repeatedly in this mode.
- \text{action}=3 — It is necessary to wrap up the session by calling cwavec one last time with \text{action}=3. This performs any housekeeping required by PV-WAVE such as closing any open files. The return value for this mode is 1. Once cwavec has been called in this mode, it should not be called again.

\text{numcmds} — The number of elements supplied in \text{cmds}. This argument is ignored if \text{action}=3 or if \text{action}=1.

\text{cmds} — An array of pointers to strings. If \text{action}=2, \text{cmds} provides an array of PV-WAVE commands to execute. This argument is ignored if \text{action}=3 or if \text{action}=1.

Returned Value

\text{istat} — The returned value depends on the \text{action} selected, as explained previously.

Discussion

You can choose to communicate with PV-WAVE in either an interactive mode or by sending an array of commands. Both of these methods automatically initialize PV-WAVE.

The first parameter is the \text{action} parameter. The action parameter may have one of the following the values:
The third parameter is the name of an array of pointers to strings (i.e., char**) containing the PV-WAVE commands to be executed. The second parameter specifies the number of elements supplied in the third parameter. The second and third parameters are ignored if the value of the action parameter is 1 or 3.

The status value returned by cwavec depends on the value of the action parameter and in some cases on the value of the action performed. If the value of the action parameter is 1 or 3, cwavec will return 1 as the status. If the value of the action parameter is 2, cwavec will return the value of the PV-WAVE system variable !Err as the status.

**Accessing the Data in PV-WAVE Variables**

The wavevars function can be used to access the results generated by PV-WAVE in a user-written application called with cwavec. wavevars is a C function that can be invoked from code linked to PV-WAVE to obtain data from PV-WAVE’s data variable space.

For detailed information on wavevars, see *Accessing Data in PV-WAVE Variables* on page 109.

**NOTE** See also the section *Using the Option Programming Interface* on page 113. The Option Programming Interface (OPI) functions allow user-written C code to access PV-WAVE variables and use other PV-WAVE functionality. OPI provides greater flexibility and control than wavevars.

**Ending the Session with PV-WAVE**

If you are in interactive mode (action=1), enter EXIT at the WAVE> prompt to return to your C application. There is no need to call cwavec with action=3 to end the session. However, if the application has accessed PV-WAVE in non-interactive mode (action=2), the session must be terminated by a final call to cwavec with action=3.

**Running PV-WAVE from a C Program**

To run PV-WAVE from a C program you must first link the C program with PV-WAVE’s Dynamic Link Library (DLL). The C program may then invoke PV-

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Run PV-WAVE interactively.</td>
</tr>
<tr>
<td>2</td>
<td>Execute a sequence of PV-WAVE commands and return to the C program.</td>
</tr>
<tr>
<td>3</td>
<td>Exit PV-WAVE and return to the C program.</td>
</tr>
</tbody>
</table>
WAVE via the entry point cwavec in the PV-WAVE DLL. The C program must pass three parameters to the cwavec entry point. A makefile is provided to link the following example programs. The makefile is available online in the same directories as the example program files. For more information, see the following example sections.

**NOTE** If you have developed previous applications linked with PV-WAVE Advantage for Windows NT 4.2, see *Console Versus Windows Subsystem Applications* on page 119 for information on modifying your makefile.

### Example 1: Execute PV-WAVE Commands From C Program

This example shows how to pass a five-element array to PV-WAVE via cwavec, have PV-WAVE perform some calculations, and produce a plot.

You can find the following listed file in:  
%WAVE_DIR%\demo\interapp\win32\cwavec\example.c

```c
#include <stdio.h>
main ()
{
/* Variables for array calculations */

   int action, numcmds, istat, cwavec ();
   char *cmds[5];

   /* Access PV-WAVE in non-interactive mode */
   action = 2;
   numcmds = 5;

   /* Send the array of commands to PV-WAVE CL */
   Define the array A
   * Perform matrix multiplication
   * Print contents of B
```
* Display B as a surface
* Issue a wait command so you can view result
* Call cwavec
*/
cmds[0] = 'a = INDGEN(5) * 4';
cmds[1] = 'b = a # a';
cmds[2] = 'PRINT, b';
cmds[3] = 'SURFACE, b';
cmds[4] = 'WAIT, 3.0';
istat = cwavec (action, numcmds, cmds);

/**
* Since we are done sending commands to
* PV-WAVE CL, we make a final call to cwavec
* with action = 3 to wrap up the session.
*/
action = 3;
istat = cwavec (action, 0, cmds);
return 0;
}

**Compiling and Linking the Example Program**

A makefile (called makefile) is provided to compile the example program and link it to the PV-WAVE DLL. This makefile is in the same directory as the source file listed above. From a Command window prompt, enter the following command to run the makefile:
nmake

**NOTE** If you have developed previous applications linked with PV-WAVE Advantage for Windows NT 4.2, see *Console Versus Windows Subsystem Applications* on page 119 for information on modifying your makefile.

**NOTE** You must have a supported Windows C compiler on your system for this makefile to execute properly.
Running the Program

From a Command window prompt, enter the following command to run the example program:

```
example
```

Cwavefor: Calling PV-WAVE from a FORTRAN Program

The cwavefor routine is the FORTRAN application entry point to a PV-WAVE DLL.

Usage

```
istat = cwavefor(action, numcmds, ptr, cmdlen)
```

Parameters

- **action** — Specifies how you wish PV-WAVE to execute. It can have one of the following values:
  - `action=1` — Run normally. You are interactively prompted for input and execution continues until you enter the end-of-file character or issue the EXIT command. At this point, cwavefor returns with a value of 1. Once cwavefor has been called in this mode, it should not be called again.
  - `action=2` — Execute the commands in the string array pointed to by `ptr`. The return value is the value of the !Error system variable. The cwavefor routine can be called repeatedly in this mode.
  - `action=3` — It is necessary to wrap up the session by calling cwavefor one last time with `action=3`. This performs any housekeeping required by PV-WAVE such as closing any open files. The return value for this mode is 1. Once cwavefor has been called in this mode, it should not be called again.

- **numcmds** — The number of elements supplied in the string array pointed to by `ptr`. This argument is ignored if `action=3` or if `action=1`.

- **ptr** — A pointer to an array of strings. If `action=2`, `ptr` provides a pointer to the array of PV-WAVE commands to execute. This argument is ignored if `action=3` or if `action=1`.

- **cmdlen** — The declared length of each string element in the two-dimensional array.

Returned Value

- **istat** — The returned value depends on the `action` selected, as explained previously.
**Discussion**

You can choose to communicate with PV-WAVE in either an interactive mode or by sending an array of commands. These methods automatically initialize PV-WAVE.

The first parameter is the *action* parameter. The action parameter may have one of the following the values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Run PV-WAVE interactively.</td>
</tr>
<tr>
<td>2</td>
<td>Execute a sequence of PV-WAVE commands and return to the FORTRAN program.</td>
</tr>
<tr>
<td>3</td>
<td>Exit PV-WAVE and return to the FORTRAN program.</td>
</tr>
</tbody>
</table>

The third parameter is the name of an array of strings containing the PV-WAVE commands to be executed. The second parameter specifies the number of elements supplied in the third parameter. The second and third parameters are ignored if the value of the *action* parameter is 1 or 3.

The status value returned by *cwavefor* depends on the value of the *action* parameter and in some cases on the value of the action performed. If the value of the *action* parameter is 1 or 3, *cwavefor* will return 1 as the status. If the value of the *action* parameter is 2, *cwavefor* will return the value of the PV-WAVE system variable !Error as the status.

**Ending the Session with PV-WAVE**

If you are in interactive mode (*action*=1), enter EXIT at the WAVE> prompt to return to your FORTRAN application. There is no need to call *cwavefor* with *action*=3 to end the session. However, if the application has accessed PV-WAVE in non-interactive mode (*action*=2), the session must be terminated by a final call to *cwavefor* with *action*=3.

**Running PV-WAVE from a FORTRAN Program**

To run PV-WAVE from a FORTRAN program you must first link the FORTRAN program with PV-WAVE. The FORTRAN program can then invoke PV-WAVE via the entry point *cwavefor* in the PV-WAVE shareable object. The FORTRAN program must pass four parameters to the *cwavefor* entry point.
**Example 1**

In non-interactive mode, valid PV-WAVE commands are passed to `cwavefor` as an array of strings. For example, to plot the vector [1, 2, 3, 4, 5] from a FORTRAN application statically linked to PV-WAVE, the commands would be:

```fortran
character *50 cmds(5)
.
.
.
cmds(1) = 'a = INDGEN(5) + 1'
cmds(2) = 'plot, a'
action = 2
ptr = LOC(cmds)
call cwavefor(action, 2, ptr, 50)
```

**Example 2**

This example shows how to pass a five-element array to PV-WAVE via `cwavefor`, have PV-WAVE perform some calculations, and produce a plot. You can find the following listed file in:

`%WAVE_DIR%\demo\interapp\win32\cwavefor\examplef.for`

**NOTE** The `MS$ATTRIBUTES` referenced in this program must appear in every FORTRAN program that is linked to PV-WAVE. For information on `MS$ATTRIBUTES`, refer to the *Fortran PowerStation Programmer’s Guide*.

```
C
C Build interface for Microsoft Fortran PowerStation (TM)
C to call the C function “CWAVEFOR” that is built into
C PV-WAVE
C
INTERFACE
INTEGER FUNCTION CWAVEFOR (action, numcmds, ptr, 
$ cmdlen)
  !MS$ATTRIBUTES C, ALIAS:’_cwavefor’ :: CWAVEFOR
  INTEGER*4 action
  !MS$ATTRIBUTES REFERENCE :: action
  INTEGER*4 numcmds
  !MS$ATTRIBUTES REFERENCE :: numcmds
  INTEGER*4 ptr
  INTEGER*4 cmdlen
  !MS$ATTRIBUTES REFERENCE :: cmdlen
```
END FUNCTION
END INTERFACE

C
C Variables for array calculations
C
integer*4  action, numcmds, cmdlen, istat
character *30 cmds(5)

C Declare pointer to pass the CHARACTER array of commands
C to PV-WAVE
C NOTE: The second part of this pointer is not used and
C Microsoft Fortran PowerStation will give a warning about this.
C
pointer (p,VAR)

C Access PV-WAVE CL in non-interactive mode
C
action = 2
numcmds = 5
cmdlen = 30
C
C Send the array of commands to PV-WAVE CL
C Define the array A
C Perform matrix multiplication
C Print contents of B
C Display B as a surface
C Issue a wait command so user can view result
C Call cwavefor
C
   cmds(1) = 'a = INDGEN(5) * 4'
   cmds(2) = 'b = a # a'
   cmds(3) = 'PRINT, b'
   cmds(4) = 'SURFACE, b'
   cmds(5) = 'WAIT, 3.0'
C
C Set up the pointer and call PV-WAVE
C
p=LOC(cmds)
   istat = cwavefor (action, numcmds, p, cmdlen)
C
C Since we are done sending commands to PV-WAVE, we make a final
C call to cwavec with action = 3 to wrap up the session.
Compiling and Linking the Example Program

NOTE This application requires Microsoft Fortran PowerStation™ 4.0.

Before compiling the application, make sure that you have set the \texttt{WAVE\_DIR} and \texttt{CPU} environment variables, as these are required by the compiler. The value of \texttt{WAVE\_DIR}, of course, depends on the location of the PV-WAVE installation at your site. For example:

\begin{verbatim}
c:> set wave_dir=\vni\wave
c:> set cpu=i386
\end{verbatim}

If necessary, source the Microsoft FORTRAN PowerStation and Microsoft Visual C/C++ setup file. Doing so will set some important environment variables and modify your path so that the compiler and linker can be found. The exact command that you enter will depend on your local configuration. For example:

\begin{verbatim}
c:> d:\msdev\bin\fpsvars
c:> d:\msdev\bin\vcvars32
\end{verbatim}

This example, \texttt{examplef.exe}, can be compiled and linked by entering the following commands:

\begin{verbatim}
c:> cd %wave_dir%\demo\interapp\win32\cwavefor
c:> f32 examplef.for %WAVE\_DIR%\bin\bin.i386nt\vniwave.lib
\end{verbatim}

Once the executable \texttt{examplef.exe} is built, you can run it by entering the name of the executable at the MS-DOS prompt.

For example:

\begin{verbatim}
c:> examplef
\end{verbatim}

\textbf{Example 3}

In this example, the FORTRAN program passes commands to PV-WAVE to be executed and then accesses the results directly (via a C wrapper) from PV-WAVE’s variable data space using the C function \texttt{wavevars}.

This example uses two functions:
• wave_from_fort.for — The FORTRAN function that calls PV-WAVE and accesses PV-WAVE variables directly.

• wavevars_fl.c — A C function (wrapper) that allows the FORTRAN program to retrieve and/or modify the values of floating-point arrays in PV-WAVE’s variable data space. This is accomplished via the wavevars function, which interacts directly with PV-WAVE’s variable data space. (Direct interaction between a FORTRAN program and wavevars does not work because FORTRAN lacks the C language’s ability to access a common data area by address.)

The C and FORTRAN code described in this example is available online in the directory:

%WAVE_DIR%/demo\interapp\win32\cwavefor

The FORTRAN program must be compiled and linked with PV-WAVE and the C wrapper routine to produce a single executable program, as explained in the next section. It is because your program is linked with PV-WAVE as a single executable that your program can share PV-WAVE variables.

Compiling and Linking the Example Program

NOTE This application requires Microsoft Fortran PowerStation™ 4.0. It also requires a C compiler. The file wavevars_fl.c was tested with Microsoft Visual C™ or Visual C++™ 4.0 and modifications may be needed if you are using a different C compiler.

This example wave_from_fort.exe can be compiled and linked by entering the following commands:

```c
set include=%INCLUDE%;%WAVE_DIR%/util\variables
cl /c wavevars_fl.c
fl32 wave_from_fort.for wavevars_fl.obj
%WAVE_DIR%/bin\bin.i386nt\vniwave.lib
```

Once the executable wave_from_fort.exe is built, you can run it by entering the name of the executable at the MS-DOS prompt. For example:

```c
wave_from_fort
```
Running the Program

After the program is compiled and linked, it can be run by entering the name of the resulting executable file. For example if the executable is called `wave_from_fort`, enter:

```
wave_from_fort
```

The output from this example is shown in Figure 2-1 on page 42.

---

Accessing Data in PV-WAVE Variables

You can access PV-WAVE variables from a dynamically linked C program by calling the function `wavevars`. Once commands have been sent to PV-WAVE from an external application, you can use the `wavevars` function to access the results in the external application. `wavevars` is a C function that can be invoked from code linked to PV-WAVE with either `cwave` or `LINKNLOAD`.

`wavevars` obtains data directly from PV-WAVE’s variable data space.

**NOTE** See also the section Using the Option Programming Interface on page 113. The Option Programming Interface (OPI) functions allow user-written C code to access PV-WAVE variables and use other PV-WAVE functionality. OPI provides greater flexibility and control than `wavevars`.

---

**Usage**

```c
int argc;
char **argv;
WaveVariable *argp;
result = wavevars(&argc, &argv, &argp);
```

**Parameters**

- `argc` — Set to the number of variables returned.
- `argv` — Set to be an array of strings, sorted in lexicographic order, corresponding to variable names available at the current scope level of PV-WAVE.
- `argp` — A type `WaveVariable` array of descriptors defining the type, structure, and dimension of the variables as well as providing a pointer to their actual data. The `WaveVariable` structure is described in the Discussion section that follows.
Returned Value

**result** — A C int value which is nonzero if the routine executed successfully, and zero if an error (such as running out of memory) occurred.

Discussion

PV-WAVE variables can be accessed directly from a C function by calling the C function `wavevars` which is dynamically linked to PV-WAVE. The C function passes three parameters to the `wavevars` entry point.

The first parameter is the address of an integer variable into which `wavevars` will return the number of currently-defined PV-WAVE variables (including system variables). The second parameter is the address of an array of pointers to strings (i.e., `char**`) into which `wavevars` will return the names of currently-defined PV-WAVE variables. The third parameter is the address of an array of pointers to the C structure `WaveVariable` into which `wavevars` will return information regarding the type, structure, dimension, and data of each PV-WAVE variable (including a pointer to the current value of the variable).

`WaveVariable` is defined as follows in:

`%WAVE_DIR%\util\variables\wavevars.h`

This header file must be included in any C function that calls `wavevars`.

```c
typedef struct WaveVariable {
    int type;
    int read_only;
    int numdims;
    int dims[8];
    int numelems;
    void **data;
    char name[MAXIDLEN + 1];
} WaveVariable;
```

**CAUTION** Although `wavevars` returns pointers to the data associated with PV-WAVE’s variables, keep in mind that the data pointer associated with a given variable can change after execution of certain PV-WAVE system commands. It’s best to call `wavevars` immediately before it is needed to obtain information from the external program.

The `wavevars` function allocates space to store the information it returns to the caller. When the caller no longer needs the information returned by `wavevars`, then the `free_wavevars()` function should be called to free the space. The
arguments to free_wavevars() should be identical to those used in the call to wavevars such as:

result = free_wavevars( &argc, &argv, &argp );

and argc must still contain the number of variables returned by the wavevars call.

The WaveVariable structure's fields are:

**int type** — The type field indicates the type of the variable. Valid PV-WAVE variable types, together with their C equivalents, are defined in wavevars.h as follows:

- TYP_BYTE char;
- TYP_INT short;
- TYP_LONG long;
- TYP_FLOAT float;
- TYP_DOUBLE double;
- TYP_COMPLEX struct { float r, i; } COMPLEX;
- TYP_STRING char *;

In PV-WAVE, a structure is a collection of data where each field (tag) has a name. The C structure WaveVariable describes a PV-WAVE structure with a type of TYP_STRUCT, where each element of the structure is contained in a list of WaveVariable structures pointed to by the data field, which is described later in this section.

The constant TYP_ARRAY will be bitwise or-ed into the type field if the variable is in fact an array.

**int read_only** — Many PV-WAVE variables are read-only, and thus if this field is nonzero, it is not permissible to alter the actual variable data. This is often the case with system variables.

**int numdims** — PV-WAVE variables may be of dimension zero (scalar) to eight. The field numdims indicates the dimensionality of the variable.

**int dims[8]** — Indicates the size of each dimension of a variable if it is of type array.

**int numelems** — Corresponds to the total number of data values which are addressable from the data pointer.

**void **data** — Corresponds to the address of the actual variable data. The data is always stored as a one-dimensional C array regardless of the dimensionality of the PV-WAVE variable.
char name[MAXIDLEN + 1] — Only used when the variable being described is of type structure and represents the structure or tag field name (depending on context).

To access a specific PV-WAVE variable you must search the array of variable names returned by wavevars to find the index associated with that variable. Then use the index to access the correct PV-WAVE variable from the WaveVariable array. The type field in WaveVariable is used to determine a variable’s type. To access the data associated with a PV-WAVE variable it is necessary to use the data pointer and cast it to the correct type. It is then possible to read and/or modify the actual data value(s).

**Using wavevars to Retrieve Data from PV-WAVE**

An example C program demonstrating the use of wavevars can be found online. You can print the program or view it online using a text editor. The example program is in the file:

```
%WAVE_DIR%\demo\interapp\win32\wavevars\example.c
```

The C program retrieves a list of all PV-WAVE variables and prints out their contents. The program demonstrates several important concepts.

- The data pointer must be cast to appropriate type.
- The data is always stored as a flat one-dimensional array.
- PV-WAVE structures are stored recursively.

**Building the DLL File**

The makefile (called makefile) creates the DLL file used by LINKNLOAD to link the C function to PV-WAVE at runtime. This makefile is in the same directory as the source file example.c. At an MS-DOS Command window prompt, enter the following command to build the DLL:

```
nmake
```

**NOTE** You must have a supported Windows C compiler on your system for this makefile to execute properly.

**Accessing the External Function with LINKNLOAD**

The following PV-WAVE procedure runs the example C program by calling LINKNLOAD. For detailed information on LINKNLOAD, see the Using LINKNLOAD to Call External Programs on page 96. This procedure is available online in the file: 

```
```
PRO variable
    ln = LINKNLOAD('example.dll','printallvars')
    INFO, ln
END

To run this procedure, start PV-WAVE and type the following command at the WAVE> prompt:

WAVE> lnl_example

The output of this PV-WAVE procedure prints a listing of all currently defined PV-WAVE variables and their values. Finally, the INFO command prints the return value of the LINKNLOAD call.

Using the Option Programming Interface

The Option Programming Interface (OPI), a C-callable or FORTRAN-callable programming interface, was developed to provide greater flexibility and control than wavevars. OPI differs from wavevars in the following ways:

• Uses less memory than wavevars
• Can obtain information about a single PV-WAVE variable at a time.
• Can obtain a subset of the information normally returned by wavevars.
• Can create new PV-WAVE variables.

For detailed information on OPI, see the PV-WAVE Programmer’s Guide.

NOTE To use OPI effectively with C programs, you should be a C programmer, understand the difference between call-by-reference and call-by-value, and be able to use pointers and the C malloc() function.

Loading PV-WAVE Dynamically into an Application

Rather than statically linking to PV-WAVE DLLs (using LIB files) when you compile, you can instead dynamically link applications to the DLLs at run time when you use cwavec(). To do this, use the LoadLibrary() and GetProcAddress() API calls to load the libraries and call into them at run time.

To accomplish these API calls, use the routine LoadWave(), located in the directory:
<wave_dir>\demo\interapp\win32\loadwave
The LoadWave() function loads PV-WAVE DLLs into your application using LoadLibrary() and calls GetProcAddress() to determine entry points for cwavec() and all of the OPI functions.

LoadWave() uses the information in the Windows Registry to find where you have installed PV-WAVE and uses this information to find the DLLs. Because it uses the Registry, LoadWave() will make your application more portable. Plus, since the DLLs are loaded and the entry points found at run time, your application will not be linked to any specific version of PV-WAVE; rather it will use whatever version is available.

In the directory shown above, you will also find an sample application that uses Microsoft Foundation Classes (MFC) to connect a PV-WAVE graphics window to a Windows application.

---

**Special Considerations for Noninteractive Applications**

If you are writing a noninteractive application — one for which no input is required at the PV-WAVE prompt — then you may need to build some special handling into your code for any nonblocking PV-WAVE widgets. You may also need to make special arrangements for graphic-window resizing.

**Using Nonblocking PV-WAVE Widgets**

When PV-WAVE's input is not coming from an interactive terminal, then the non-blocking Widget event loop (WwLoop /NoBlock, for example) may not behave as expected. This event loop is normally serviced as PV-WAVE "watches" the keyboard. Thus, when you do not have an interactive terminal, the event loop is not serviced.

Some examples of noninteractive applications include:
- Running commands from within a PV-WAVE procedure or batch file using @ or .RUN
- Sending commands to PV-WAVE via a pipe using wavecmd()
- Running PV-WAVE in batch mode or redirecting standard input
- Using cwavec() or cwavefor() (see the Note, below)

In these situations, it becomes the programmer’s responsibility to service the event loop. If you are running commands in a procedure file, then this can be as simple as exiting back to the command prompt.
If you are running PV-WAVE as a background process, however, or in some mode where you need a nonblocking event loop but do not have an interactive terminal, then you will need call \texttt{WtProcessEvent} periodically to service the event loop. To process all pending events, you call this function using the \texttt{/Drain} keyword.

\textbf{Example}

Say you have the following in a procedure file:

\begin{verbatim}
NAVIGATOR
  ; A non-blocking Widget application
  WHILE ( WtProcessEvent(/Drain) NE 1 ) DO BEGIN $
    PRINT, 'Still alive' & $
    WAIT, 0.1
    PRINT, 'Exiting'
  END
EXIT
\end{verbatim}

When you execute this procedure at the PV-WAVE prompt, the \texttt{WHILE} loop causes \texttt{WtProcessEvent} to be called until the widget application is exited. Of course, in a real case, you would do some other processing rather than just \texttt{PRINT} and \texttt{WAIT}, but the point is that you need to call \texttt{WtProcessEvent} periodically and check its return status (so you know when the Widget application is finished).

\textbf{NOTE} When you use \texttt{cwavec()} and \texttt{cwavefor()}, the normal behavior for \texttt{WwLoop, /NoBlock} is to ignore the keyword and actually block. So if you call \texttt{cwavec()} to execute a nonblocking Widget, PV-WAVE will service its own event loop until the Widget exits. If you wish to have the event loop nonblocking, then you must call \texttt{WtProcessEvent} periodically to process the events and use \texttt{WwLoop, NoBlock=2} to force the nonblocking behavior.

\textbf{Window Resizing}

PV-WAVE will not automatically recognize resizing of graphic windows when it is not being run interactively (for example, using \texttt{wavecmd()} or \texttt{cwavec()}). To notify PV-WAVE of window resizing use:

\begin{verbatim}
WSET, winid
\end{verbatim}

This command has the effect of updating the \texttt{!D} system variable. See \texttt{WSET} for more information.
Using Dynamic Data Exchange (DDE)

This section assumes that you are already familiar with DDE, its terminology, and use. The terms client (destination application), server (source application), conversation, service, topic, item, data, and manual link used in this section are standard terms related to DDE. If you are not familiar with DDE, refer to a Microsoft Windows user’s guide.

Overview of DDE Support in PV-WAVE

DDE allows communication between applications running in the Windows environment. Using DDE, you can establish a conversation between PV-WAVE and another application that supports DDE. Currently, you can establish PV-WAVE as a server application that is called by a client application; however, you cannot establish PV-WAVE as a client.

In general, you can perform the following tasks using DDE with PV-WAVE:

- **Setting variables** — The client application can set variables in PV-WAVE to specified values. The variable must already exist in PV-WAVE before it can be set.

- **Querying variables** — The client application requests that PV-WAVE send it the value of a specified variable.

- **Executing commands** — Any PV-WAVE command can be executed from the client application. For example, after an application processes data, the data is sent to PV-WAVE followed by graphics commands to display it.

Initializing PV-WAVE as a DDE Server

You must start the PV-WAVE DDE server before you can use DDE to exchange data with other applications. All DDE communication is then passed through the DDE callback function.

To start the PV-WAVE DDE server, enter the following command

```
wavedde
```

or click on the **PV-WAVE DDE Server** icon in the PV-WAVE Program Group window on Windows NT, or use the **Start** button from Windows 95.

Textual output and messages are displayed in the shell window from which the PV-WAVE DDE Server was launched. If the server was launched from an icon or the **Start** button, then a separate console window is created on the desktop to display the output.
Which DDE Topics Are Available?
The PV-WAVE server’s service name is \texttt{WAVE}. The PV-WAVE server responds to the topics CONNECT, DISCONNECT, EXECUTE, PUT, and GET. Each of these topics is discussed in the following sections.

\textbf{CONNECT}
Establishes communication from the client to the PV-WAVE server.

\textbf{DISCONNECT}
Closes communication between the client application and the PV-WAVE DDE server.

\textbf{EXECUTE}
The EXECUTE topic lets the client application send a string containing a PV-WAVE command to be executed. The string cannot start with a dollar ($$) or period (\.) character. The EXECUTE topic does not have an item associated with it.

\textbf{PUT}
Use the PUT topic when you want to modify the value of a variable in PV-WAVE. The variable must already exist in PV-WAVE and be of the expected size or an error results.

The PUT topic’s item is always a string containing the name of the variable being set. The item’s data is a CF_TEXT formatted string.

\textbf{GET}
Use the GET topic when you want to query the value of variable in PV-WAVE. The variable must already exist in PV-WAVE or an error results.

The GET topic’s item is always a string containing the name of the variable being queried. The GET topic returns data representing the result of the query, and this data is “poked” into the client application.

Which Data Formats Are Available?
The PV-WAVE DDE server supports only the CF_TEXT data format. Each text record ends with a CR-LF combination, and a NULL character signals the end of the data.
Communicating with a Windows Application Written in C

This section discusses an example Windows application written in C. The application allows the user to interact directly with PV-WAVE. More specifically, the application is a simple client designed to initiate DDE conversations with PV-WAVE that:

- execute PV-WAVE commands
- set PV-WAVE variables
- query PV-WAVE variables

Each conversation is established manually, which means that PV-WAVE only supplies data to the C application when the C application requests it.

The Example C Program

The example C program is provided online in the file:

%WAVE_DIR%\demo\interapp\win32\dde\ddetest.c

The C application executes PV-WAVE commands, sets the values of existing variables, and queries the values of existing variables. It uses the topics GET, PUT, and EXECUTE.

A makefile is available in the same directory as the C file ddetest.c. To compile this application, enter the following command from an MS-DOS Command window prompt:

nmake

**NOTE** You must have a supported Windows C compiler on your system for this makefile to execute properly.

Running the Example Application

To run the application, you must first start the PV-WAVE DDE server as explained previously.

To start the application, enter the following command at the operating system prompt:

ddetest
The following dialog box appears on your screen:

![DDE Wave Tester dialog box](image)

**Figure 3-1** The dialog box for the client application. This simple application uses DDE to communicate with PV-WAVE.

To execute a PV-WAVE from the C application, type the command in the text field next to the **Execute** button, and then click **Execute**.

To create a variable in PV-WAVE, enter an expression in the text field next to the **Execute** button and click **Execute**. For example, to create a four element array variable called `numbers`, you could enter:

```
numbers = [1,2,3,4]
```

and then click **Execute**.

To change the value of this variable, type the variable’s name in the **Variable** text field, then enter a new value in the **Query/Set** text field and click **Set**. To see the value of a variable, enter its name in the **Variable** text field and click **Query**.

---

**Console Versus Windows Subsystem Applications**

You may use PV-WAVE within your application in one of two ways:

- The “PV-WAVE Console” (*wave.exe*)
- The “PV-WAVE Home Window” (*wavewin2.exe*).

The *wave.exe* executable is built as a console-mode application, whereas *wavewin2.exe* is a Windows application.

The libraries provided in the PV-WAVE DLLs are built as Windows Subsystems and thus will support either type of application you wish to write (using `cwavec()`, for example).
PV-Wave As a Console Application

As a console application, the `wave.exe` command supports standard I/O redirection and can be used in batch files. Pipes for standard I/O are also supported, and you can run PV-WAVE over a network (using telnet or remote login, for example) for non-graphical applications.

When you run `wave.exe` from either from the Program Manager or from a Windows 95 Explorer icon, a console window is automatically created to facilitate I/O.

PV-Wave As a Home Window Application

The PV-WAVE “Home Window” (`wavewin2.exe`) is a Windows application. Therefore, it always runs in its own window. This allows more flexibility in communicating to other applications — the Clipboard, for example. However, this type of application does not support standard I/O redirection and cannot be run from a batch file or across a network.

Sample Applications are Available

For sample programs and makefiles demonstrating both console- or Windows-subsystem applications, look in the directory:

(Windows) `<wavedir>\demo\interapp\win32`

Where `<wavedir>` is the main PV-WAVE directory.

Additional Documentation

Consult the *Win32 Tool User’s Guide* and the *Win32 Programmer’s Reference* if you have any question about the linker, changes to the linker flags, or the initial application entry points.
Building VDA Tools

Read this chapter if you intend to develop VDA Tools or if you wish to customize or modify existing VDA Tools.

What Are VDA Tools?

VDA Tools represent a new paradigm in PV-WAVE application development.

On the surface, a VDA Tool is a PV-WAVE application with a Motif or Microsoft Windows Graphical User Interface (GUI) (see Figure 4-1). VDA Tools resemble WAVE Widgets applications; however, VDA Tools provide a level of flexibility and independence that distinguishes them from typical WAVE Widgets applications.

This section lists the main features of VDA Tools. Subsequent sections describe VDA Tools on a more technical level and discuss how to build VDA Tools. The final section describes how you can create a Navigator interface for one or more VDA Tools.
Multiple Instances from Same Source Code

Consider the VDA Tool WzPlot. The source code resides in one file: wzplot.pro. As with any VDA Tool, you can call WzPlot as many times as you wish. Each time you call WzPlot, a new WzPlot window is created on your screen. Each of these windows is a separate instance of this VDA Tool. Each instance can display a different variable. In one instance, data is plotted in red, in another, data is green. One instance might have a legend, and another might have callout lines with associated text used to highlight a portion of the data.

Each instance of a VDA Tool is completely independent from all other instances of that same tool, even though they share the same source code. The VDA Tools Manager, described later, enables this VDA Tool independence. As a VDA Tool developer, you will use VDA Tools Manager Application Programming Interface (API) calls in your VDA Tool code to communicate with the Tools Manager.
Intertool Communication

Although multiple instances of VDA Tools are independent, the Tools Manager permits communication between tools. For example, you can export a variable from one tool to another or cut and paste graphical elements — rectangles, lines, legends, axes, bitmaps, and text — from one tool to another. For information on graphical elements, see *Manipulating Graphical Elements on page 136*.

In addition, data selection information is shared among VDA Tools. In other words, the selected portion of a variable is displayed as selected in every VDA Tool currently displaying that variable.

Called from the Command-line

Each VDA Tool can be run directly from the PV-WAVE command line. In addition, VDA Tools can be run directly from an interface called a Navigator.

Non-blocking

VDA Tools do not block the PV-WAVE command line, nor do they block one another.

Code Generation

If you are developing PV-WAVE applications, you can use VDA Tools to generate PV-WAVE code. The code generation feature allows the user to write the PV-WAVE commands used to produce a VDA Tool result (plot data, import data, etc.) to a file.

Easy to Save and Restore

A save function is provided to let you save the entire contents of a VDA Tool, which you can restore later. You can also save a template. A template is a “blank” VDA Tool. It includes the entire VDA Tool except for the data. In other words, you can restore a VDA Tool view, all of the color setups, annotations, and other modifications you have made, without restoring the data.

Portable

VDA Tools developed for an X Windows environment (under UNIX or OpenVMS) are completely portable to Microsoft Windows. Likewise, VDA Tools developed
under Windows can be run in an X Windows environment (under UNIX or OpenVMS).

**Native Look and Feel**

In an X Windows environment, VDA Tools adopt the Motif interface conventions. Under Microsoft Windows, VDA Tools look and behave just like typical Windows applications.

**Easy to Build and Customize**

PV-WAVE comes with a set of prewritten VDA Tools. By convention, the commands used to start these Tools all begin with the letters “Wz”, such as WzPlot and WzSurface. You can customize any of these existing VDA Tools, or build your own VDA Tools. You are free to use the existing VDA Tool routines as templates and examples.

**Online Help**

Each VDA Tool provided with PV-WAVE comes with complete online Help. Select the Help menu on the right-hand side of the VDA Tool menu bar.

---

**A Technical Perspective on VDA Tools**

The previous section presented a general overview of VDA Tools. The next few sections discuss VDA Tools on a more technical level. These sections are the starting point for anyone who plans to design, build, or customize VDA Tools. The next few sections cover the following topics:

- VDA Tool Ingredients
- VDA Tools Manager
- VDA Tools Utilities
- Navigator

*Figure 4-2* shows the overall architecture of a VDA Tool.
Figure 4-2 VDA Tool architecture.

*Figure 4-3* shows the directories in which the predefined VDA Tool (Wz) routines, VDA Tool Manager (Tm) routines, VDA Tool Utilities (Wo) routines, and their resource files are located.
What is the VDA Tools Manager?

The VDA Tools Manager is a software layer that keeps track of VDA Tools, assists in saving and restoring VDA Tools, and facilitates the communication between VDA Tools.

The Tools Manager contains a data structure that stores the following VDA Tool information:

- **Unique names of VDA Tools** — The unique name is the key to the data structure. Tools Manager API calls allow you to update and modify this structure based on the unique name. The function TmGetUniqueToolName is used to generate a unique name for each VDA Tool instance.

- **Methods** — A method is a program that is executed in response to a trigger in a VDA Tool, such as a menu or button selection or mouse click. Normally, a VDA Tool will have several methods defined for it.

- **Graphical Elements** — Graphical elements (GRAELS) are predefined graphics routines used by VDA Tools. Graphical elements are primarily used for annotation — lines, rectangles, legends, text — but also include axes.
• **Variables** — The Tools Manager keeps track of which variables are associated with specific VDA Tools. The Tools Manager layer allows variables to be passed back and forth between different VDA Tools.

• **Attributes and values** — Anything that is tracked by the Tools Manager can have attributes assigned to it. For example, color is an attribute that might be assigned to a variety of items, such as lines, text, variables, and axes. The attribute’s value determines how the item is realized in the VDA Tool. For example, the thickness attribute for a line might have a value of 2 pixels.

• **Save/Restore Information** — The information needed to save and restore a VDA Tool is always maintained by the Tools Manager.

The Tools Manager relies on a central data structure that maintains necessary information about each instance of a VDA Tool. The VDA Tools Manager API routines, which all begin with “Tm”, allow VDA Tools to update this structure and retrieve information from it.

The VDA Tools Manager enables VDA Tool independence. As a VDA Tool developer, you will use Tools Manager API calls in your VDA Tool code to communicate with the Tools Manager. *Figure 4-4* shows how multiple VDA Tool instances share the same source code, which communicates with the Tools Manager core.
A VDA Tool, such as WzPlot, is a GUI application that consists of graphical elements, variables, methods, and attributes. All instances of a VDA Tool share the same code. The Tools Manager keeps track of the specific characteristics of each instance of a VDA Tool. The VDA Tool source program communicates with the Tools Manager via the Tools Manager API (Tm*) routines.

**Figure 4-4**
**The Tools Manager API**

The routines used for creating and manipulating VDA Tools are listed in the following table.

**Functional Listing of Tools Manager General Routines**

<table>
<thead>
<tr>
<th>Types of Routines</th>
<th>Routine Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Routines</td>
<td>TmDynamicDisplay</td>
</tr>
<tr>
<td></td>
<td>TmEnumerateItems</td>
</tr>
<tr>
<td></td>
<td>TmEnumerateToolNames</td>
</tr>
<tr>
<td></td>
<td>TmGetMessage</td>
</tr>
<tr>
<td></td>
<td>TmGetTop</td>
</tr>
<tr>
<td></td>
<td>TmGetUniqueToolName</td>
</tr>
<tr>
<td></td>
<td>TmInit</td>
</tr>
<tr>
<td></td>
<td>TmRegister</td>
</tr>
<tr>
<td></td>
<td>TmUnregister</td>
</tr>
<tr>
<td>Attribute Routines</td>
<td>TmEnumerateAttributes</td>
</tr>
<tr>
<td></td>
<td>TmGetAttribute</td>
</tr>
<tr>
<td></td>
<td>TmSetAttribute</td>
</tr>
<tr>
<td>Method Routines</td>
<td>TmEnumerateMethods</td>
</tr>
<tr>
<td></td>
<td>TmExecuteMethod</td>
</tr>
<tr>
<td></td>
<td>TmGetMethod</td>
</tr>
<tr>
<td></td>
<td>TmSetMethod</td>
</tr>
<tr>
<td>Variable Routines</td>
<td>TmAddVar</td>
</tr>
<tr>
<td></td>
<td>TmDelVar</td>
</tr>
<tr>
<td></td>
<td>TmDynamicShowVars</td>
</tr>
<tr>
<td></td>
<td>TmEnumerateVars</td>
</tr>
<tr>
<td></td>
<td>TmGetVarMainName</td>
</tr>
<tr>
<td>Save/Restore Routines</td>
<td>TmRestoreTemplate</td>
</tr>
<tr>
<td></td>
<td>TmRestoreTools</td>
</tr>
<tr>
<td></td>
<td>TmSaveTools</td>
</tr>
<tr>
<td>Intertool Communication Routines</td>
<td>TmAddSelectedVars</td>
</tr>
<tr>
<td></td>
<td>TmDeselectVars</td>
</tr>
<tr>
<td></td>
<td>TmEnumerateSelectedVars</td>
</tr>
<tr>
<td></td>
<td>TmExport</td>
</tr>
<tr>
<td></td>
<td>TmExportSelection</td>
</tr>
</tbody>
</table>
In general, a VDA Tool is a collection of the following parts, which are managed by the VDA Tools Manager:

- Unique name
- Methods
- Variables
- Graphical elements
- Attributes
- User interface

PV-WAVE underlies the entire VDA Tool architecture. All VDA Tool code is written in PV-WAVE.

**VDA Tools Require Unique Names**

Once initialized with the TmInit command, the Tools Manager data structure is ready to keep track of VDA Tools. The key to this tracking system is the VDA Tool name. Each VDA Tool must be registered with the Tools Manager with a unique name. A unique name for a VDA Tool is obtained via a call to TmGetUniqueTool-
Names. The TmRegister function is then used to register the unique name with the Tools Manager.

Each time a VDA Tool is called, it is registered with the Tools Manager with a unique name. For example, the first instance of WzScatterPlot might be registered as WzScatterPlot_0, and the second instance might be WzScatterPlot_1.

You can always determine the names of all VDA Tool instances currently registered with the tools manager with the TmEnumerateToolNames function. The following example command shows that three separate instances of WzScatterPlot are currently registered with the Tools Manager.

```
PRINT, TmEnumerateToolNames()

  WzScatterPlot_0, WzScatterPlot_1, WzScatterPlot_2
```

**Methods Drive VDA Tools**

A method is a procedure that is executed in response to a trigger in a VDA Tool, such as a menu or button selection or mouse click. Normally, a VDA Tool will have several methods defined for it.

When a method is executed, the Tools Manager helps direct the subsequent action. The Tools Manager keeps track of the unique instance of the VDA Tool to apply the method to, the name of the program to execute, and any data associated with that specific tool that the method procedure needs.

**Register the Method**

First, the method name and method procedure name must be registered with the VDA Tools Manager (see Figure 4-5). This is usually done somewhere in the main VDA Tool procedure. The TmSetMethod function accomplishes this:

```
TmSetMethod, tool_name, 'TM_DISPLAY', 'WzPlotDisplay'
```

where `tool_name` is the unique name of a VDA Tool, `TM_DISPLAY` is the name of the method, and `WzPlotDisplay` is the name of the method procedure (the procedure that is executed when the method is triggered).

**Execute the Method**

Next, a method is executed by a TmExecuteMethod call. The VDA Tool program determines when a method is executed — this is up to the VDA Tools developer. Some predefined triggers are built into the VDA Utilities routines WoMenuBar and WoButtonBox.

```
TmExecuteMethod, tool_name, 'TM_DISPLAY'
```
Events that can trigger methods include: selecting a menu item, selecting a button from the tool bar, clicking a mouse button when the pointer is in the drawing area, exposing a window, and, potentially, many others. Figure 4-5 illustrates the sequence of actions that lead to the execution of a method.

(1) TmSetMethod, unique_name, ‘TM_VIEWATTR’, $ 'WzPlotViewAttr'

(2) View Attributes

(3) TmExecuteMethod, Toolname, $ 'TM_VIEWATTR'

(4) WzPlotViewAttr

(5) View Attributes Dialog Box

**Figure 4-5** Executing a method requires the following basic sequence of actions: (1) The TM_VIEWATTR method is registered with the method procedure WzPlotViewAttr in the VDA Tool source code. (2) The View Attributes menu item is selected. (3) This triggers a TmExecuteMethod command. (4) The method procedure, WzPlotViewAttr, is executed. (5) The method procedure displays the View Attributes Dialog Box.

**Standard Methods**

The following table lists and describes the set of “standard” VDA Tool methods. These method names are considered to be standard because they are called by default by some of the VDA Utilities (Wo) routines. Technically, you can change the names of the methods to anything you like, but you might also need to modify the callbacks for some of the VDA Utilities routines as well.
NOTE As a VDA Tools developer, you can create any methods you need to develop the functionality for a specific VDA Tool. For instance, if you are developing a table tool, you might not need to use the TM_DISPLAY method at all, and you would need to develop new methods designed to work with tables.

<table>
<thead>
<tr>
<th>Method</th>
<th>Action</th>
<th>Called by</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM_DISPLAY</td>
<td>Performs the graphics commands to display data.</td>
<td>Menu commands (redraw), action area callbacks, graphical element routines (redraw), window manager event (expose a window).</td>
</tr>
<tr>
<td>TM_CODEGEN</td>
<td>Writes the PV-WAVE code used to display the graphics to a file.</td>
<td>Menu command callback.</td>
</tr>
<tr>
<td>TM_CONVERT</td>
<td>Performs coordinate system conversions for copy and paste operations between windows with different coordinate systems or aspect ratios.</td>
<td>Data export, resize, and graphical element routines.</td>
</tr>
<tr>
<td>TM_VIEWATTRIBUTES</td>
<td>Creates a user interface (dialog box) for setting view attributes.</td>
<td>Menu command callbacks and graphical element routines.</td>
</tr>
<tr>
<td>TM_DATA_EXPORT</td>
<td>Creates a user interface (dialog box) for exporting a variable from the VDA Tool.</td>
<td>Menu command callback.</td>
</tr>
<tr>
<td>TM_DATA_SELECTION</td>
<td>Verifies that data was selected. This method is triggered when a data selection is completed. You can select data by clicking on it or drawing a rectangle around it. Calls TmDynamicDisplay.</td>
<td>Menu command and button bar callbacks, graphical element routines, from drawing area callback.</td>
</tr>
<tr>
<td>TM_VARATTRIBUTES</td>
<td>Creates a user interface (dialog box) for setting variable attributes.</td>
<td>Menu command callbacks and graphical element routines.</td>
</tr>
<tr>
<td>TM_DATA_IMPORT</td>
<td>Checks the validity of data being imported into a VDA Tool.</td>
<td>The target VDA Tool when data is exported to it.</td>
</tr>
</tbody>
</table>
Deciding which items need to be defined for a VDA Tool, along with the attributes of each item, is one of the challenges of VDA Tool development.

Items can include a wide variety of things that you want to put in a VDA Tool. Some examples of items are:

- variables
- help files
- drawing area
- file names

Each item can have characteristics, called attributes, associated with it.

Different VDA Tools might have very different sets of items, depending on the purpose of the VDA Tool. Once an item is defined and registered with the Tools Manager, the Tools Manager can always keep track of that item’s characteristics for a specific VDA Tool instance.

For example, if you are developing a surface plotting tool, you might want to provide a way for the user to change the rotation of the axes. Note that whenever the surface tool draws a surface plot, it needs to be able to extract the information from the Tools Manager that is needed to render the orientation (that is, if anything but the default rotation is to be used).

The following calls to the TmSetAttribute function assign the attributes \texttt{X\_ROTATION} and \texttt{Z\_ROTATION} to the global item for the tool, which is called \texttt{TM}. These attributes are assigned the default values of 30.

\textbf{NOTE} These calls can be placed in the main procedure of the VDA Tool to set defaults, and they can be in the procedure used to create a View Attributes dialog box.

\begin{verbatim}
  tmp=TmSetAttribute(tool_name, 'TM', 'X\_ROTATION', 30)
  tmp=TmSetAttribute(tool_name, 'TM', 'Z\_ROTATION', 30)
\end{verbatim}

With these orientation items registered in the Tools Manager for the item \texttt{TM} and unique Tool instance \texttt{tool\_name}, the procedure associated with a method, such as \texttt{TM\_DESTROY} allows clean-up of anything created during the execution of the VDA Tool. For instance, temporary variables can be removed.
as the display method, can get the settings it needs to display the surface correctly. For example, the procedure called by the TM_DISPLAY method for this VDA Tool might contain the following calls:

\[
\text{XRot} = \text{TmGetAttribute(tool\_name, 'TM', 'X\_ROTATION')} \\
\text{ZRot} = \text{TmGetAttribute(tool\_name, 'TM', 'Z\_ROTATION')} \\
\]

Then, the returned values from these calls can be used directly in the SURFACE procedure to draw the surface:

\[
\text{SURFACE, var1, Ax=XRot, Az=ZRot} \\
\]

The following table summarizes the “standard” items and their attributes:

<table>
<thead>
<tr>
<th>Item</th>
<th>Attribute</th>
<th>Purpose</th>
<th>Called by</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM_HELP</td>
<td>ON_WINDOW</td>
<td>Access a Help topic</td>
<td>Help callback (on window)</td>
</tr>
<tr>
<td>TM_HELP</td>
<td>HELP_FILE</td>
<td>Access a Help topic</td>
<td>Help callback</td>
</tr>
<tr>
<td>TM_HELP</td>
<td>ON_VERSION</td>
<td>Access a Help topic</td>
<td>Help callback (version)</td>
</tr>
<tr>
<td>TM_WINDOWID</td>
<td>0 (Window 0)</td>
<td>PV-WAVE Window ID</td>
<td>Grael routines</td>
</tr>
<tr>
<td>TM_DRAWING</td>
<td>0 (Window 0)</td>
<td>WAVE Widgets Drawing Area handle</td>
<td>Tools Manager handlers for the Drawing Area</td>
</tr>
<tr>
<td>TM IMPORT</td>
<td>IMPORT</td>
<td>Accept or do not accept imported variables</td>
<td>View Attributes dialog box</td>
</tr>
<tr>
<td>TM REPLACEVAR</td>
<td>REPLACEVAR</td>
<td>Replace existing variables with imported variables, or add imported variables to the VDA Tool.</td>
<td>View Attributes dialog box</td>
</tr>
<tr>
<td>TM XRANGE</td>
<td>XRANGE</td>
<td>Set the range of the (x)-axis graphical element.</td>
<td>View Attributes dialog box</td>
</tr>
<tr>
<td>TM YRANGE</td>
<td>YRANGE</td>
<td>Set the range of the (y)-axis graphical element.</td>
<td>View Attributes dialog box</td>
</tr>
<tr>
<td>TM SELECTED_DATA</td>
<td>SELECTED_DATA</td>
<td>Sets the selection mode: rectangular area or single point.</td>
<td>Data selection button or menu callback.</td>
</tr>
</tbody>
</table>
The important things to remember about variables, attributes, and values are:

- You must register variables (or other items) and their attributes and values with the Tools Manager.
- You can use Tools Manager API calls to modify or extract item attributes and values when they are needed.
- You can define any items you need to provide specific functionality in a VDA Tool.
- Variables and other items have attributes and values associated with them, which you also define.
- Each unique VDA Tool instance has its own set of variables or other items associated with it.

### Manipulating Graphical Elements

Graphical elements (also referred to as Graels) are predefined graphics routines used by VDA Tools. These routines allow you to easily add, configure, and remove graphical elements in a VDA Tool display area. The graphical elements are equivalent to items associated with specific VDA Tool instances. Graphical elements allow the VDA Tool user to draw the following graphics and text items on the fly:

- Rectangle
- Line
- Legend
- Text
- Axis
- Bitmap

This standard set of Graels is accessible from the standard menu bar and button bar that is provided with the VDA Utility routines WoMenuBar and WoButtonBar.
NOTE For the most part, VDA Tools developers do not need to be concerned with how graphical elements are created. The prewritten graphical element routines handle all aspects of drawing these graphics objects — positioning, scaling, coordinate conversion, as well as the actual drawing. Functions on the standard graphical menu and button bar have working callbacks that call the graphical element routines directly. When the user adds a graphical element to a VDA Tool plot, it is automatically registered as an item with the Tools Manager.

To obtain the names of the graphical elements that are currently registered with a VDA Tool, use the TmEnumerateGraels function.

\[ \text{items} = \text{TmEnumerateGraels} (\text{tool\_name}) \]

where \textit{tool\_name} is the unique name of a VDA Tool instance. This function returns an array of strings containing the names of the items associated with that particular VDA Tool. Graels are returned in the following format:

\[ \text{graelname\_num} \]

For example, the following call shows that the VDA Tool has two variables registered with it, and several Graels: two rectangles, a line, and a legend.

\[ \text{PRINT, TmEnumerateGraels(tool\_name)} \]
\[ \quad \text{var1 var2 rectangle\_0 rectangle\_1 line\_0 legend\_0} \]

Once you know the names of the Graels that are registered with a VDA Tool, you can use other Tools Manager routines to extract and modify their attributes and values.

For example, the following line of code obtains the names of several items, three of which are Graels: two lines and a rectangle.

\[ \text{PRINT, TmEnumerateGraels(tool\_name)} \]
\[ \quad \text{var1 var2 rectangle\_0 line\_0 line\_2} \]

These Graels probably have attributes, such as color, linestyle, and line thickness. You can retrieve the attributes with the TmEnumerateAttributes function. The following lines of code show that the Grael \text{LINE\_0} has three attributes associated with it: color, linestyle, and thickness.

\[ \text{PRINT, TmEnumerateAttributes(tool\_name, 'LINE\_0')} \]
\[ \quad \text{COLOR LINESTYLE THICKNESS} \]

This information can then be used to modify the Grael \text{LINE\_0}. It is up to the VDA Tools developer to determine how the user can modify any item. For a Grael, you might design a View Attributes dialog box in which the user can change the
characteristics of a selected Grael. The TmSetAttribute function provides the means of accomplishing this:

tmp = TmSetAttribute(tool_name, ‘LINE_0’, ‘COLOR’, value)

where value is a user-supplied value for the plot color. This value might have been obtained from a View Attributes dialog box.

**Graphical Element API Routines**

The following table lists the routines used to manage graphical elements, which are primarily used to annotate plots — lines, rectangles, text, axes, and legends. They are used primarily if you want to add new graphical elements or modify existing ones, which, in general, is not necessary.

Functional Listing of Tools Manager Graphical Routines

<table>
<thead>
<tr>
<th>Types of Grael Routines</th>
<th>Grael Routine Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Grael Routines</td>
<td>TmAddGrael</td>
</tr>
<tr>
<td></td>
<td>TmAxis</td>
</tr>
<tr>
<td></td>
<td>TmBitmap</td>
</tr>
<tr>
<td></td>
<td>TmDelGrael</td>
</tr>
<tr>
<td></td>
<td>TmEnumerateGraels</td>
</tr>
<tr>
<td></td>
<td>TmGetGraelRectangle</td>
</tr>
<tr>
<td></td>
<td>TmGetUniqueGraelName</td>
</tr>
<tr>
<td></td>
<td>TmLegend</td>
</tr>
<tr>
<td></td>
<td>TmLine</td>
</tr>
<tr>
<td></td>
<td>TmRect</td>
</tr>
<tr>
<td></td>
<td>TmSetGraelRectangle</td>
</tr>
<tr>
<td></td>
<td>TmText</td>
</tr>
<tr>
<td>Grael Method Routines</td>
<td>TmEnumerateGraelMethods</td>
</tr>
<tr>
<td></td>
<td>TmExecuteGraelMethod</td>
</tr>
<tr>
<td></td>
<td>TmGetGraelMethod</td>
</tr>
<tr>
<td></td>
<td>TmSetGraelMethod</td>
</tr>
<tr>
<td>Grael Selection Routines</td>
<td>TmAddSelectedGrael</td>
</tr>
<tr>
<td></td>
<td>TmDelSelectedGraels</td>
</tr>
<tr>
<td></td>
<td>TmEnumerateSelectedGraels</td>
</tr>
<tr>
<td>Grael Z-Order Routines</td>
<td>TmBottomGrael</td>
</tr>
<tr>
<td></td>
<td>TmTopGrael</td>
</tr>
<tr>
<td>Grael Grouping Routines</td>
<td>TmGroupGraels</td>
</tr>
<tr>
<td></td>
<td>TmUngroupGraels</td>
</tr>
</tbody>
</table>
User Interface

The user interface for a VDA Tool consists of a top-level layout WAVE Widget and a collection of child widgets such as menus, buttons, drawing areas, and text areas. WAVE Widgets are described in Chapter 5, *Using WAVE Widgets*.  

**NOTE** The VDA Utilities are a set of convenience routines that provide high-level compound widgets for use in VDA Tools, including a menu bar, button bar, dialog boxes, and text area. These functions and their API are described in *VDA Utilities* on page 139.

---

**VDA Utilities**

VDA Utilities are a set of high-level convenience routines that developers can use to quickly develop and modify the user interface of a VDA Tool. These routines are written in PV-WAVE and are located in:

- **(UNIX)** `<wavedir>/lib/vdatools`
- **(OpenVMS)** `<wavedir>: [LIB.VDATOOLS]`
- **(Windows)** `<wavedir>\lib\vdatools`

Where `<wavedir>` is the main PV-WAVE directory.

VDA Utilities all begin with the prefix “Wo”.

The VDA Utilities layer allows the application developer to establish a consistent and native application look and feel, have readily portable code, and take advantage of inherent international support, while retaining some flexibility in terms of user interface design.

The calls to VDA Utilities are the same on all platforms: applications based on VDA Utilities for their user interface are immediately portable among all supported UNIX, OpenVMS, and Windows platforms.

---

Functional Listing of Tools Manager Graphical Routines (Continued)

<table>
<thead>
<tr>
<th>Types of Grael Routines</th>
<th>Grael Routine Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grael Cut, Copy, Paste, and</td>
<td>TmCopy</td>
</tr>
<tr>
<td>Delete Routines</td>
<td>TmCut</td>
</tr>
<tr>
<td></td>
<td>TmDelete</td>
</tr>
<tr>
<td></td>
<td>TmPaste</td>
</tr>
</tbody>
</table>

---
VDA Utilities take advantage of PV-WAVE string and resource database facilities, which allow you to adapt your applications easily to different languages.

Finally, VDA Utilities offer user interface flexibility. You can easily add new menu items and functions to the standard menu and button bar utilities.

Three important VDA Utilities are:

- WoMenuBar
- WoButtonBar
- WoMessageArea

These three VDA Utilities are used to create the menus, buttons, and message area in the graphical VDA Tools — WzPlot, WzImage, WzContour, and WzSurface.

## Connecting to Online Help

The set of VDA Tools provided with PV-WAVE feature a context sensitive online Help system. On UNIX and OpenVMS systems, the online Help that is used is Hyperhelp from Bristol Technology. On Windows systems, the native Winhelp system is used.

To access an online Help topic, a VDA Tool needs to know the name of the Help file and the name of the specific topic to display.

The method TM_HELP activates the Help system and displays a specified topic in the viewer.

See also *Adding Online Help on page 152.*

## Example

The following two calls set attributes that can be retrieved and used in the PV-WAVE HELP command to call a particular online Help topic.

```c
old_f = TmSetAttribute(tool_name, 'TM_HELP', 'HELP_FILE', $<filepath>)
; This call sets an attribute — the filepath for an online Help file.

old_t = TmSetAttribute(tool_name, 'TM_HELP', 'TOPIC', $<topic_name>)
; This call sets an attribute — the name of an online Help topic.
```
TIP  The WoGenericDialog function accepts these two attribute settings with the Help keyword and handles calling the HELP command.

If you do not use WoGenericDialog, then you will need calls like the following to display an online Help topic:

```plaintext
file = TmGetAttribute(tool_name, 'TM_HELP', 'HELP_FILE')
; Retrieve the filepath of the online Help file.

topic = TmGetAttribute(tool_name, 'TM_HELP', 'TOPIC')
; Retrieve the Help topic name.

HELP, topic, Filename=file
; Display the help topic.
```

---

**Tool-to-Tool Communication Routines**

VDA Tools can exchange data such as variables, graphical elements, and selection information. For variables, an export function is provided on the standard File menu. This export function lets you export a variable to one or more VDA Tools. For graphical elements, functions on the standard Edit menu and the Button Bar let you cut or copy graphical elements from one VDA Tool and paste them in another VDA Tool. Data that is selected on one VDA Tool is shown highlighted in all other active VDA Tools displaying the same variable.

**The Selection List**

The Tools Manager uses a selection list to exchange data between VDA Tools. First, data to be exported or copied is placed on a “selected list”. The TmAddSelectedVars and TmAddSelectedGrael routines accomplish this step. The TmDeselectVars and TmDelSelectedGraels command removes specified variables or graphical elements from the selected list.

The TmExportSelection or TmPaste command can be used to send the data or graphical element(s) currently on the selection list to another VDA Tool.

**Exporting Variables from $MAIN$**

VDA Tools rely on the TmExport function to export variables on the $MAIN$ program level to specified VDA Tool(s).
VDA Tools Can Accept or Reject Exported Variables

You can provide a function so that the user can select whether or not a VDA Tool will accept an exported variable. This function is available on the Attributes=>View Attributes dialog box of graphical VDA Tools such as WzPlot.

The following calls to TmSetAttribute set whether or not VDA Tools will accept exported variables.

```pascal
old = TmSetAttribute(tool_name, 'TM', 'IMPORT', 1)
    ; Sets the IMPORT attribute to 1. This setting indicates that the VDA Tool will accept variables.

old = TmSetAttribute(tool_name, 'TM', 'IMPORT', 0)
    ; Sets the IMPORT attribute to 0. This setting indicates that the VDA Tool will accept not variables.

old = TmSetAttribute(tool_name, 'TM', 'REPLACEVAR', 1)
    ; Sets the REPLACEVAR attribute to 1. This setting indicates that the VDA Tool will replace the current variable instead of adding to its list of variables.
```

The TM_DATA_IMPORT method must be defined for the given VDA Tool. The method procedure for TM_DATA_IMPORT must perform the appropriate import actions.

We recommend you use the TM_DATA_IMPORT method procedure to check the value of the IMPORT attribute, and display an appropriate alert if importing is disabled.

Creating Optional Areas in VDA Tools

You can make specific areas in a VDA Tool optional and provide a menu that allows the user to show or hide these optional areas. For example, the WzSurface VDA Tool contains three optional areas: the Buttonbar, the Controls area, and the Message area. Using the Options menu, the user can choose to hide or show any combination of these areas.

The following sections discuss the steps required to create optional areas in VDA Tools.
Modify the Menu Structure

Add the Option menu code to the menu structure for the VDA Tool. To do this, edit the VDA Tool menu file. For instance, for WzSurface, this file is called wzsurfacemenus.pro.

The Option menu code, shown below, should be inserted immediately before the code for the Windows menu:

```
NAME: ['OptionsMenu', 'OptionsMenu'],$
MENUBUTTON: '',$
MENU:{,CALLBACK:'WoGMBOptionsButtonBarCB',$
    NAME: 'OptionsButtonBar',$
    TOGGLE: '',$
    CALLBACK:'WoGMBOptionsControlsAreaCB', $
    NAME: 'OptionsControlsArea',$
    TOGGLE: '',$
    CALLBACK:'WoGMBOptionsMessageAreaCB', $
    NAME: 'OptionsMessageArea',$
    TOGGLE: ''$
},$
```

Adjust the Menu Pane Number

For existing VDA Tools, in the main VDA Tool procedure file (e.g., wzsurface.pro), increment by one the pane number that is saved for the WINDOW_MENU attribute.

For example,
```
tmp = TmSetAttribute(tool_name, 'MENUBAR', 'WINDOW_MENU', 6)
```
should be changed to:
```
tmp = TmSetAttribute(tool_name, 'MENUBAR', 'WINDOW_MENU', 7)
```

Set the Status of the Option Menu Toggles

Set the status of the menu toggles to the correct state by getting the current toggle settings and resetting them accordingly. This step is needed because it is possible that these values were saved in a previous session and then restored.

The following code fragment is taken from wzsurface.pro. It shows how TmGetAttribute is used to get the current status (shown or hidden) for the Button-
bar, Controls area, and Message area. The returned values are then used in
WoMenuBarSetToggle to reset the Option menu toggles.

\[
\text{status} = \{ \\
TmGetAttribute(tool\_name, ‘TM’, ‘BUTTONBAR\_STATUS’, Default=1), \\
TmGetAttribute(tool\_name, ‘TM’, ‘CONTROLS\_STATUS’, Default=1), \\
TmGetAttribute(tool\_name, ‘TM’, ‘MESSAGE\_STATUS’, Default=1) \\
\}
\]

WoMenuBarSetToggle, tool\_name, [7, 7, 7], [1, 2, 3], status

**Call WoGMBOptionsInit**

After creating all the optional areas in the VDA Tool (e.g., for WzSurface, this
includes the Buttonbar, Controls area, and Message area), call the routine
WoGMBOptionsInit:

WoGMBOptionsInit, tool\_name

This routine checks the *\_STATUS attributes and hides those areas that should not
appear when the VDA Tool is managed.

**Implementation for Optional Message Area**

The WoMessage routine is equipped to automatically handle the unregistering of
the message area.

**Implementation for Optional Button Bar**

There are two requirements for creating an optional button bar. First, the button bar
and the status area (tool tips) must be in a single form that can be unmanaged. Sec-
ond, the parent of the form must also be a form with attachments to the other
widgets in the tool.

For example, if the original code is:

bblayout = WwLayout(layout, /Form, Top=bar, /Left, /Right)
tb = WoButtonBar(bblayout, tool\_name, /Top, /Graphics, $ 
/Left, /Right)

The following modifications must be made, where the button bar and status area
are children of a form, which is in turn the child of a form.
bbtoplayout = WwLayout(layout, /Form, Top=bar, /Left, /Right)
    Create a top-level form.

bblayout = WwLayout(bbtoplayout, /Form, /Top, /Left, /Right)
    Create a child form that can be unmanaged.

tb = WoButtonBar(bblayout, tool_name, /Top, /Graphics, $
    /Left, /Right)
    Attach the button bar to bblayout.

s = TmSetAttribute(tool_name, 'TM', 'BUTTONBAR_ID', bblayout)
    Attach the status area to bblayout.

All attachments from other widgets must be made to bbtoplayout. Specifically
the drawing area and controls areas should have a top attachment to
bbtoplayout.

Implementation for Optional Controls Area

An optional controls area is created much like the optional button bar. First, the
controls area must be in a layout that can be unmanaged. Second, like with the but-
ton bar, the parent of the form must also be a form with attachments to the other
widgets in the tool.

The following lines of code demonstrate the correct way to create an optional con-
trols area:

ctltopLayout = WwLayout(layout, /Form, Top=bbtoplayout, $
    Bottom=ms, /Left)
    Create the top-level layout.

controlsLayout = WwLayout(ctltopLayout, /Form, /Frame, $
    /Top, /Bottom, /Left, /Right)
    Create a child form that can be unmanaged.

save_wid = WwGetValue(controlsLayout, /Parent)

While it looks like the WwGetValue call returns ctltopLayout it really returns the
widget ID for the frame widget that was created. If the second WwLayout call
above did not have the Frame keyword, the call to WwGetValue would not have
been necessary, and the controlsLayout widget ID could be used in the following
assignment statement.

s = TmSetAttribute(tool_name, 'TM', 'CONTROLS_ID', save_wid)
Creating a Navigator

A Navigator is an interface for VDA Tools. For instance, a Navigator can be created to combine several VDA Tools into a single application. A Navigator is provided with PV-WAVE; however, an application programmer can easily modify this pre-written Navigator or create an entirely new Navigator.

The PV-WAVE Navigator

The PV-WAVE Navigator is a prewritten application that is provided with PV-WAVE. To run the PV-WAVE Navigator, enter Navigator at the PV-WAVE command line.

The PV-WAVE Navigator is located in the directory shown in Figure 4-6:

```
Main VNI Product Installation Area (VNI_DIR)

   navigator-1.0  wave  license  <options>

   lib

   navmenus.pro
   navtools.pro
   navigator.pro
   navigator.cpr
```

Figure 4-6  PV-WAVE Navigator directory structure.

PV-WAVE Navigator Files

The PV-WAVE Navigator consists of the following three files.

- `navmenus.pro` — Contains definitions for the menus on the Navigator and specifies the menu item callbacks.
- `navtools.pro` — Contains a list of the names of the VDA Tools, the number of variables that each VDA Tool can accept, and the position coordinates of each VDA Tool in the Navigator.
• navigator.pro — This is the main program file for the Navigator. It contains the callback procedures for all of the callbacks in navmenus.pro. This file contains routines that handle session saving and restoring. In addition, this file contains the main startup routine for the Navigator.

If you create your own Navigator, we recommend that you follow this basic design; however, you might not need all the components of the PV-WAVE Navigator. For instance, you could create a Navigator that does not have a menu bar.

Modifying the PV-WAVE Navigator

You can modify the existing PV-WAVE Navigator in a number of ways. For instance, you can add or remove VDA Tools from the Navigator.

To modify the Navigator, edit the appropriate files in the navigator-1_0 directory. (We recommend that you make a copy of any file you change.)

For example, to remove the WzSurface Tool from the Navigator, edit the file navtools.pro to remove the WzSurface name, number of variables that the VDA Tool can accept, and position elements. You do not need to modify the navigator.pro file to add or remove a VDA Tool.

TIP You can also use the Configure menu on the Navigator to add or remove VDA Tools. The Configure menu provides an interactive way to perform the same task described in this section.

Creating a New Navigator

The most basic requirements for a Navigator are that it must be able to call VDA Tools and determine which variable is currently selected. The second requirement is important so that the Navigator knows what variable to load when a particular VDA Tool is called.

The PV-WAVE Navigator is only one possible way in which a Navigator could be configured. Depending on the application you have in mind, you may want to create an entirely different Navigator.

Step 1 Decide what tools you want your Navigator to have.

Step 2 Decide what kind of user interface you want. You might choose not to have a menu bar, for instance.
Step 3  Write the main Navigator procedures. To do this, we recommend that you look at the existing PV-WAVE Navigator files for guidance. These files define menus and buttons, specify the VDA Tools in the Navigator, implement callbacks, and initialize the Navigator.

Resources for the Navigator

The user interface resources for the PV-WAVE Navigator are located in the same area as the other VDA Tool resources as shown in Figure 4-7:

```
Main PV-WAVE Directory (WAVE_DIR)
   ├── lib
   │   └── vdatools
   │       └── wz*.pro
   │           └── wo*.pro
   │               └── tm*.pro
   └── xres
       ├── american
       │     └── vdatools
       │        └── nav.ad
       │            └── nav.ads
       └── other_lang
            └── vdatools
                 └── nav.ad
                     └── nav.ads
```

Figure 4-7  Directory tree showing the location of X Resources for the Navigator and VDA Tools.

Resources and Strings

GUI application management is aided by using resource and string files. Resource and string files provide an easy method for customizing and internationalizing PV-WAVE applications. This is accomplished by isolating the dynamic aspects of the application in files which are distinct from the code. Thus, the ability to customize becomes a process of translating a few files in tight proximity rather than many individual lines of code spread throughout the application. Resources are distinguished from strings because they describe the appearance or behavior of widgets, while strings are not specifically widget-related.
Resources

Using resources to describe the appearance of the application enables you to customize Wave Widgets or VDA Tools to your applications, without needing to modify the code. In addition, PV-WAVE is designed to allow multiple resource files per application, so the various resources can be loaded as needed rather than loading everything at the application startup.

A resource definition as it appears in a resource file is really a name/value pair. The name, the first part of the pair, describes the widget to which the resource value is applied. For instance, the following resource name/value pair:

```
myapp.save_button.labelString: Save
```

essentially says,

Within the application `myapp` there is a button named `save_button`. Set the label on that button to the string "Save".

Resource files are a collection of many resource name/value definitions, such as the previous example, that are loaded as needed prior to creating the widgets they describe.

Resource names are constructed by traversing the widget tree to the target widget instance. The precise path required depends on the design of your particular application. Widgets are named explicitly using specific keywords found in the Wave Widget commands. The Ww layer routines you use to create a widget have one or more of the following keywords: `Name`, `Layout_Name` or `Shell_Name`. These keywords use one or more resource datatype strings to create widgets with the specified names.

Resources loaded using the PV-WAVE resource-loading routines (see Using Resources and Strings in PV-WAVE Applications on page 150) are merged into the standard system resource database which is used when widgets are created. As a result, no special commands are necessary to assign the resource values to a widget; instead, this is done automatically when the widget is created.

**NOTE**  You must name the resource so that it matches the name of the widget it modifies.

Strings

Strings are similar to resources in that they allow you to customize your PV-WAVE applications without having to modify the code. The difference is that strings do not directly describe the appearance or behavior of a widget like resources do. Strings are often used for error messages or to construct labels that are completed during the execution of the application.

For instance, suppose the application opens a user-specified file as part of its commands. The application should have the capability to display an error message in the event the named file doesn’t exist when the application is run. Since it’s desirable to specifically name the file as part of the message, the message construction should consist of a static string (such as 'Cannot open the file') and a dynamic file name. String files are used for just this purpose.

The string file syntax is very similar to that of a resource file; it’s comprised of a name/value pair. Unlike resource definitions, however, the name portion of the string definition is a single identifying string rather than a hierarchical definition. For instance, the following string definition:

```
myapp_open_error: Cannot open the file
```

defines a string suitable for the example.

Strings loaded using the PV-WAVE string-manipulating routines are loaded into a common string database. The database is distinct from the resource database, so no interaction between strings and resources is possible.

**NOTE** Special care should be taken to ensure that the name used in the string definition is unique among all other strings loaded by any application.

A good way to ensure your string names are unique is to add the application name to the front of the string name. In this way, the possibility of conflicting definitions exists only if two applications have the same name.

**Using Resources and Strings in PV-WAVE Applications**

The resource handling routines and the corresponding architecture levels are shown in the following table.
The STRLOOKUP and WtResource routines form the foundation for the VDA Tool Utility functions (TmGetMessage, WoLoadResources, and WoLoadStrings). The lower-level routines require specific path and file names to access the resource and string files. The higher-level functions can be used with or without a specified pathname.

In addition to being able to use more than one resource file in a given application, you can also access two types of resource files: widgets (object-related), and strings (object-less).

**Widget Resource Files**

Widget resource files are distinguished by the .ad filename extension. These files allow you to customize the appearance and behavior of parent and compound widgets created using WAVE Widget routines. This custom naming feature, in turn, improves the maintainability of GUI applications.

The widget initialization routines WwInit and WtInit allow you to specify a Resource keyword. This keyword specifies a resource file to load. If this keyword is not used, PV-WAVE looks for the following file by default:

- **(UNIX)** $WAVE_DIR/bin/Wave.ad
- **(OpenVMS)** WAVE_DIR:[BIN]WAVE.AD

Note that the Wave.ad file is not distributed with PV-WAVE. You must supply this file if you would like PV-WAVE to load it.

You can load resources at any time after calling WtInit or WwInit. For WAVE Widgets applications, the routine WwResource is used to load, add, or save additional resources. VDA Tools use the utility routine, WoLoadResources procedure, to load resource files. Prior to loading the resource file, WoLoadResources searches directory paths to find the resource file to load.
String Files

String resource files contain error messages and other text strings, and are recognized by their .ads filename extension. The capability to specify and access multiple object-less resources allows you to customize messages and other text used in your application, including internationalization of your applications.

String files may be loaded at any time in a PV-WAVE session, since the STRLOOKUP function doesn’t require the Wave Widgets environment to be initialized. Once a string file has been loaded into the string database, STRLOOKUP may be used to extract values.

VDA Tools are designed to need only the TmGetMessage function for their string operations. Since the string file is loaded (if necessary) prior to querying the database, the filename is specified in each call to TmGetMessage. Internally, TmGetMessage uses the WoLoadStrings procedure to load the string file.

Adding Online Help

This section discusses how online help for VDA Tools is implemented. Read this section if you plan to incorporate online help into your own VDA Tools applications. Two methods of creating online help for VDA Tools are discussed. One method requires that you purchase additional software needed to create help files. The second method involves using a WAVE Widgets text widget to display online help information.

If you are creating a new VDA Tools application, you may want to develop online help information for your users. On UNIX and OpenVMS platforms, PV-WAVE online help files are created with a Hyperhelp® compiler from Bristol Technology. To create the same style of online help for new VDA Tools applications, you must purchase this compiler from Bristol Technology. The compiler converts document files (in FrameMaker MIF or RTF format) to a Hyperhelp help file that can be displayed in the Hyperhelp viewer, which is provided with PV-WAVE.

On Windows platforms, online help for VDA Tools is distributed in WinHelp format. Many third-party products are available for creating WinHelp files.

NOTE Because the online help files that are distributed with PV-WAVE are binary files, there is no way to edit or modify them.

If you plan to use Hyperhelp to create new online help files for UNIX and OpenVMS platforms, you can contact Bristol Technology, Inc. about purchasing
the software required to create Hyperhelp online help files. The documentation that comes with Hyperhelp software explains how to author help files, compile them, and integrate them into your application.

Bristol Technology, Inc.
39 Old Ridgebury Road
Danbury, CT 06810-5113 USA
Phone: 203-798-1007
Fax: 203-789-1008
Email: info@bristol.com
Web: www.bristol.com

If you do not wish to use Hyperhelp or WinHelp for your customized online help system, it is possible to develop online information in ASCII format and display it using a WAVE Widget, such as the widget WwText. For detailed information on using WAVE Widgets and on WwText, see Chapter 5, Using WAVE Widgets.

Online Help Implementation Overview

A user can access online help from a VDA Tool in two ways: from the Help menu or by clicking the Help button in a dialog box. In each case, menu and dialog box button callbacks are used to activate the online help system and display the appropriate topic in the help viewer.

**NOTE** If you are using the Hyperhelp or WinHelp to develop your own online help topics, then most of the applications programming work is already done for you. Existing VDA Tool convenience routines for adding menus and dialog boxes to VDA Tools provide prewritten callback routines that activate the appropriate online help system. All you need to do when creating your own VDA Tools is to set the values for the help filename and the name of specific topic you want to display in your VDA Tool code.

**The Help Menu**

The standard VDA Tool Help menu contains four buttons:

- **On Window** — When this button is selected, the help system starts and a specific help topic is displayed. This help topic is the table of contents topic for information on that specific VDA Tool. You must supply the specific help filename and topic name in the menu callback. This callback calls the PV-WAVE HELP procedure with the *Filename* and *Keyword* parameters.
• **Index** — When this button is selected, the help system starts and displays the help index. The callback routine calls the PV-WAVE HELP procedure with the `Index` keyword.

• **On Help** — When this button is selected, the help system starts and displays information on how to use the online help system. The callback routine calls the PV-WAVE HELP procedure with the `Help` keyword.

• **On Version** — When this button is selected, the help system starts and a specific help topic is displayed. This help topic contains information on the current software version. You must supply the specific help filename and topic name in the menu callback. This callback calls the PV-WAVE HELP procedure with the `Filename` and `Keyword` parameters.

Menu callbacks are defined in `wographicsbuttonscp.pro`. In the following code fragment, note that the objective of the callback is to call the HELP routine with the proper `Filename` and `Keyword` parameters. The values of these parameters are obtained via calls to `TmGetAttribute`. The attributes themselves must have been defined for the TM_HELP method in the specific VDA Tool program.

```pro
;--------------------------------------------------------------------------
;  Help menu Callbacks
;--------------------------------------------------------------------------
; Help->On Window...
;
; -----------------------
PRO WoGMBHelpOnWindowCB, wid, index
    DECLARE FUNC, TmGetAttribute
    tool_name = WoGMBGetMenuBarToolName(wid)
    topic = TmGetAttribute(tool_name, 'TM_HELP', 'ON_WINDOW', $ Default='HELP')
    helpfile = TmGetAttribute(tool_name, 'TM_HELP', $ 'HELP_FILE', Default=GETENV('WAVE_HLPFILE'))
    HELP, topic, Filename=helpfile
END
```

The values needed by the menu callback to display the proper help file and topic are defined in the VDA Tool program using calls to `TmSetAttribute`. For example:

```pro
old_f = TmSetAttribute(tool_name, 'TM_HELP', 'HELP_FILE', $ <filepath>)
; This call sets an attribute — the filepath for an online Help file.
old_t = TmSetAttribute(tool_name, 'TM_HELP', 'TOPIC', <topic_name>)
; This call sets an attribute — the name of an online Help topic.
```
NOTE Two attributes that you must set in the VDA Tool program are called ON_WINDOW and ON_VERSION. These are the help topics that can vary for each VDA Tool and application. In general, there is no reason to change the other two menu buttons, Index and On Help, as these display topics that do not vary from application to application.

Dialog Boxes

The WoGenericDialog box is a convenience routine used extensively in VDA Tools to create dialog boxes. This routine has a Help keyword that is used to specify the help filename and topic name to use when the user clicks on the Help button. In your VDA Tool code, you need to provide the appropriate values for this keyword. WoGenericDialog then simply calls HELP with these topic and filename values.

If you do not use WoGenericDialog, then you will need calls like the following to display an online Help topic (i.e., you need to include a specific call to HELP in the callback routine for the Help button of each dialog box).

```vba
file = TmGetAttribute(tool_name, 'TM_HELP', 'HELP_FILE')
; Retrieve the filepath of the online Help file.

topic = TmGetAttribute(tool_name, 'TM_HELP', 'TOPIC')
; Retrieve the Help topic name.

HELP, topic, Filename=file
; Display the help topic.
```

Adding Online Help without Hyperhelp or WinHelp Software

If wish to add context sensitive help to a VDA Tool, but you do not wish to purchase Hyperhelp or WinHelp software, you can do so using an alternate mechanism, such as the WAVE Widget WwText. If you use WwText, you will need to author the online help information in ASCII files.

The Help Menu

If you are using the standard VDA Tool menu, as provided by the routine WoMenuBar, you will need to do two things:

- Modify the menu callbacks in wographicsbuttonsacb.pro for each menu item. You must create new callbacks because the existing callbacks are tailored specifically for Hyperhelp. The new callback you write will call the
WAVE Widgets function WwText to display the help information you have written.

- Modify the CALLBACK field in the menu structure for your Help menu. The menu structures for VDA Tools are in wz*menus.pro (for example, wzplotmenus.pro). You simply need to add the name of the callback routine you write containing the WwText call to this field.

**Dialog Box Help**

Use the WAVE Widgets WwGenericDialog, instead of WoGenericDialog, to create the dialog boxes in your VDA Tools. WoGenericDialog is set up specifically to access Hyperhelp. It would be difficult to modify this convenience routine to work with an alternative help system.

With WwGenericDialog, you can use the optional labels parameter to specify the buttons that will appear in a dialog box. The callback routine, then, can be a simple case statement that takes appropriate action depending on which button was selected. For the **Help** button, this case statement can call WwText directly to display the appropriate help information.
Using WAVE Widgets

This chapter explains how to use WAVE Widgets to create graphical user interface (GUI) applications in PV-WAVE.

The first section of this chapter is designed to help you choose the most appropriate method for your application.

The rest of this chapter describes WAVE Widgets, a set of easy-to-use, high-level widget routines that allow you to develop Motif or Windows GUIs with PV-WAVE. The Widget Toolbox and VDA Tools are discussed in the following chapters.

NOTE Visual Numerics does not support the use of WAVE Widgets with Open-VMS 24-bit display hardware.

Methods of GUI Programming in PV-WAVE

WAVE Widgets and the Widget Toolbox

WAVE Widgets is an easy-to-use set of PV-WAVE functions for creating Motif or Windows GUIs for PV-WAVE applications. Applications created with WAVE Widgets are completely portable between Motif and Windows environments. WAVE Widgets are designed for developers who require cross-platform GUI applications. See Introduction to WAVE Widgets on page 161 for more information.

The Widget Toolbox is a set of PV-WAVE functions used to create Motif or Windows Graphical User Interfaces (GUIs) for PV-WAVE applications. The Widget
Toolbox (Wt) functions call Motif or Windows GUI routines directly, and are designed primarily for developers who are already experienced with Motif or Windows GUI development. See Chapter 6, Using the Widget Toolbox, for detailed information on the Widget Toolbox functions.

Figure 5-1 shows how WAVE Widgets and the Widget Toolbox are layered on top of the Motif toolkit, Xt Intrinsics, Xlib, and the operating system.

**Figure 5-1** WAVE Widgets and the Widget Toolbox are built on top of the Motif toolkit. The Widget Toolbox also communicates with Xt Intrinsics, Xlib, and the Operating System.

**Windows Advanced Controls Library**

The Windows Advanced Controls Library (WAC) was developed to allow portability between Motif-based and Microsoft Windows applications. The WAC is built upon Windows GDI, Windows Controls, and ported subsets of X Windows and Xt Intrinsics. Above the WAC are Widget Toolbox and WAVE Widgets functions, which are designed to be portable between Microsoft Windows and X Windows.
Figure 5-2 shows the general configuration of WAVE Widgets, the Widget Toolbox, and the Windows Advanced Controls Library.

![Diagram of WAVE Widgets, Widget Toolbox, and Windows Advanced Controls Library]

Figure 5-2 The Windows Advanced Controls Library (WAC) relies on a ported subset of Xt Intrinsics and X Windows. This provides portability of the WAVE Widgets and Widget Toolbox functions between Microsoft Windows and X Window based systems.

The following table lists the advantages and disadvantages of all the methods available to developers for creating GUI applications. Use this table to help determine which method is best for you.

### GUI Development Methods for PV-WAVE

<table>
<thead>
<tr>
<th>Method</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Widget Toolbox</td>
<td>Creates Motif or Windows GUIs. All programming done in PV-WAVE. All Motif widget classes are supported. Allows rapid prototyping. Highly flexible. No knowledge of C required.</td>
<td>Application developer must be experienced using the Motif widget toolkit. Not portable between Motif Microsoft Windows environments.</td>
</tr>
<tr>
<td>Method</td>
<td>Advantages</td>
<td>Disadvantages</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>WAVE Widgets</td>
<td>Easy to use. Creates Motif or Microsoft Windows GUIs. Completely portable between Motif and Microsoft Windows environments (no modification required). All programming done in PV-WAVE. No experience programming with the Motif widget toolkit required. No knowledge of C required. Users can create new and modified WAVE Widgets functions. Allows rapid prototyping. Contains sufficient functionality for most GUI development projects.</td>
<td>Limited number of widget classes are supported. Less overall flexibility in interface design.</td>
</tr>
<tr>
<td>VDA Tools</td>
<td>All programming done in PV-WAVE. Provides easy-to-call user interface components which can reduce overall programming time. Multiple instances of a VDA Tool are instantiated from the same source code. Provides code generation functions. Easy to internationalize (via a string server).</td>
<td>Must be purchased separately. Requires substantial PV-WAVE programming experience to develop new VDA Tools not already provided. Less overall flexibility in interface design (comparable to WAVE Widgets). Limited number of widget classes are supported (comparable to WAVE Widgets).</td>
</tr>
<tr>
<td>C-based applications that call PV-WAVE</td>
<td>Highly flexible. Can be C code generated by any GUI builder. Can be used to add Visual Data Analysis capability to an existing application.</td>
<td>Application developer must be experienced using the Motif or Windows widget toolkit. Most complicated method of application development. Knowledge of C required. Knowledge of interapplication communication required.</td>
</tr>
</tbody>
</table>

(The application interface can be developed in C, and, via PV-WAVE’s interapplication communication functions, the C application can call PV-WAVE to perform data processing and display functions.)
Introduction to WAVE Widgets

WAVE Widgets provides an easy way for PV-WAVE application developers to create Motif and Windows GUIs. You can think of a widget as a user-interface object, such as a dialog box, a button box, or a file selection box. WAVE Widgets is a set of functions that create a number of different kinds of widgets. Widget characteristics such as text, color, and position are controlled using keywords. See Creating and Arranging WAVE Widgets on page 165 for a complete list of the types of WAVE Widgets.

For example, Figure 5-3 shows a Motif-style dialog box created with the WAVE Widgets function WwDialog.

![Motif-style dialog box, created with the WwDialog function.](image)

Who Uses WAVE Widgets

WAVE Widgets are designed for developers with little or no experience with Motif or Windows GUI development. Developers who are experienced GUI programers may want to use WAVE Widgets for the rapid development of application prototypes.

WAVE Widgets are Standard Library Functions

The WAVE Widgets functions are located in the PV-WAVE Standard Library in the subdirectory:

- (UNIX) `$VNI_DIR/wave/lib/std/motif`
- (OpenVMS) `VNI_DIR:[WAVE.LIB.STD.MOTIF]`
- (Windows) `%VNI_DIR%\wave\lib\std\windows`

Where VNI_DIR is the main Visual Numerics directory.
Designing Your Own WAVE Widgets

The implementation of WAVE Widgets is straightforward, making it relatively easy for developers to customize and invent their own WAVE Widgets routines. Developers who create their own WAVE Widgets routines may share them with other users by submitting them to the Users’ Library. For more information see the section the section *The Users’ Library* in Chapter 9 of the *PV-WAVE Programmer’s Guide*.

WAVE Widgets are Portable

Applications developed with WAVE Widgets are completely portable between Motif and Windows systems. No matter which window system your application is destined to use, the WAVE Widgets programming interface is always the same.

---

**First Example and Basic Steps**

This section briefly introduces the basic steps involved in creating a WAVE Widgets application.

**First Example**

The following example incorporates each of the basic steps described later in this section. To run this example, enter the callback procedure below in a file, and compile it with the .RUN command. Then enter the widget commands at the WAVE> prompt. A radio button box appears on the screen. Each time you click on a radio button, the callback procedure is executed and some information is printed in the main PV-WAVE window. The radio box is shown in *Figure 5-4*.

To dismiss the radio box widget, select the appropriate function (such as Close) from the widget’s window manager menu.
Callback Procedure

Callback procedures are routines that are executed in response to an event that occurs inside a widget, such as a mouse click.

```plaintext
PRO RadioCB, wid, which
CASE which OF
  1: PRINT,'First Toggle Selected'
  2: PRINT,'Second Toggle Selected'
  3: PRINT,'Third Toggle Selected'
ENDCASE
value = WwGetValue(wid)
PRINT, value
END
```

Widget Commands

```plaintext
top=WwInit('ww_ex1', 'Examples', layout)
labels=['System','Owner','Group']
rbox=WwRadioBox(layout,labels, 'RadioCB', $ /
  /Vertical, Border=2, Spacing=20)
status=WwSetValue(top, /Display)
WwLoop
```

The Basic Steps

To use WAVE Widgets in an application, you always follow these basic steps. These steps are described in more detail in the remainder of this chapter.

- Create callback procedures.
Callback procedures are routines that are executed in response to an event, such as clicking a button or dismissing a dialog box.

- Initialize WAVE Widgets with the WwInit function. For example:
  
  ```
  top = WwInit('appl', 'Appl', layout)
  ```

- Create widgets by calling the appropriate WAVE Widgets functions. All WAVE Widgets function names begin with Ww. For example:
  
  ```
  bbox=WwButtonBox(layout, labels, 'ButtonCB',
  /Horizontal, Spacing=20)
  ```

- Display the top-level widget with the WwSetValue command and the `Display` keyword. For example:
  
  ```
  status=WwSetValue(top, /Display)
  ```

- Execute the WwLoop function. This function executes the “event loop”, which handles events (such as mouse clicks) and dispatches callbacks (routines that are executed in response to events).

---

**Initializing WAVE Widgets**

The WAVE Widgets function WwInit initializes WAVE Widgets. You must place a call to WwInit at the beginning of any application that uses WAVE Widgets. WwInit does the following:

- Establishes a connection to the X Windows server.
- Initializes Xt Intrinsics.
- Initializes the Motif Toolkit (or the Windows Advanced Controls (WAC) Library under Windows).
- Initializes WAVE Widgets.
- Creates a top-level shell.
- Creates a layout widget inside the top-level shell.

WwInit has the following form:

```
top = WwInit(name, class, workarea [keywords])
```

The `name` parameter specifies the name of the application, and the `class` parameter indicates a more general category of application class. WwInit also creates the first top-level shell or root window and an initial layout widget. The ID of the layout widget is returned by the `workarea` parameter. The function returns the ID of the top-level shell. For example:
top=WwInit('simple_image','Examples',layout, $
    \text{Background} = \text{Skyblue}')

where \text{simple\_image} is the name of the application, and \text{Examples} specifies a general class to which \text{simple\_image} belongs. The \text{layout} parameter returns the ID of the layout widget that is created inside the top-level shell. Finally, a background color is specified with the \text{Background} keyword. The ID of the top-level shell is returned in the variable \text{top}. For more information on the top-level shell, see \textit{The Widget Hierarchy on page 166.}

One purpose of specifying a general class of application (the \text{class} parameter) is that resources can be shared, via a resource file, among elements of that class. In general, WAVE Widgets applications do not require a resource file. Some developers, however, may create applications that use both WAVE Widgets and Widget Toolbox calls to produce the GUI. In this case, a resource file can be shared by all of the widgets used in the application. Note, however, that if color keywords, such as \text{Background} and \text{Foreground}, are used in a WAVE Widgets call, the specified color(s) override color specifications made in a resource file.

For more information on layout widgets, see \textit{Arranging Widgets in a Layout on page 169.}

\textbf{Example}

Here’s a simple example showing the use of WwInit in the creation of a multi-line text widget. You can display the text widget by entering the commands as shown at the WAVE> prompt.

\begin{verbatim}
    top=WwInit('ww_ex2', 'Examples', layout)
    filename = getenv('WAVE_DIR')+'/'+Tips
    text=WwText(layout, 'TextCB', /Read, $
        File=filename, Cols=40, Rows=20)
    status=WwSetValue(top, /Display)
    WwLoop
\end{verbatim}

\textbf{Creating and Arranging WAVE Widgets}

This section explains the widget hierarchy, lists the general types of widgets that are available, and explains how to arrange widgets in a layout.
The Widget Hierarchy

WAVE Widgets applications consist of a hierarchy of widgets. The widget hierarchy refers to the top-level or root widget and all of the widgets that are related to it. The relationship between widgets in a hierarchy is usually described as a “parent/child” relationship. Each time you create a new widget, you must specify its parent.

At the top of every widget hierarchy must be a special type of widget called the root or main window. This window is created by the WwInit function. The root window widget provides an interface between the widget hierarchy and the window manager. In addition, WwInit creates a “layout” widget, which is like a container in which other widgets are arranged.

NOTE You can create additional main windows with the WwMainWindow function.

Figure 5-5 shows a WAVE Widgets GUI that is composed of a root window, two layout widgets, and ten other widgets (w 1 – w 10). These widgets could be buttons, sliders, menu bars, etc.

Figure 5-6 shows the hierarchical relationship between the widgets in the above GUI. The root window is the top-level window, and it is the parent of layout 1 (both the root window and the first layout widget are created by WwInit). Layout 1 is the parent of widgets w 1, w 2, w 3, w 4, w 5, and layout 2. Layout 2 is the parent of widgets w 6, w 7, w 8, w 9, and w 10.

Figure 5-5 Schematic of WAVE Widgets GUI that contains two layout widgets and ten other widgets.
Each separate widget is represented externally by a widget ID (a variable of type long), returned by the creation function, such as WwButtonBox.

**Types of WAVE Widgets**

The following is a list of the kinds of widgets you can create with WAVE Widgets. Examples are shown throughout this chapter.

- **Button Box** — A horizontally or vertically oriented box containing push buttons. See *Creating a Button Box and a Tool Box* on page 179 for more information.

- **Command Widget** — A widget used for command entry with a built-in command history mechanism. It includes an input text field, a label, and a command history list. See *Creating a Command Widget* on page 198 for more information.

- **Controls Box** — Horizontally or vertically oriented sliders, which can optionally contain text input fields for entering exact values. A slider allows the user to set or display the values of the variables that fall within a predefined range. See *Creating a Controls Box with Sliders* on page 182 for more information.

- **Dialog Box** — A blocking (modal) or nonblocking (modeless) dialog box containing a text input field and button box with control buttons. See *Creating Dialog Boxes* on page 193 for more information.

- **Drawing Box** — A box that displays PV-WAVE graphics. See *Creating a Drawing Area* on page 183 for more information.
• **File Selection Box** — Displays the contents of directories and lets the user select files. See *Creating a File Selection Widget on page 196* for more information.

• **Layout Widget** — A “container” used to hold other widgets in a specific arrangement. Types of arrangements include: row/column, form, and bulletin board. Keywords are used to select the type of layout and the orientation, spacing, and sizing of the widgets in the layout. By default, a layout widget is created when WAVE Widgets is initialized with the WwInit command. See *Creating and Arranging WAVE Widgets on page 165* for more information.

• **List Box** — A scrolling list that allows users to select one or more items from a group of choices using the mouse. An additional callback can be defined for the default action, activated with a double-click. See *Creating a Scrolling List on page 189* for more information.

• **Main Window** — A top-level (window manager) window and layout widget. By default, the WwInit function creates a top-level widget and a layout widget. WwMainWindow lets you create additional top-level and layout widgets.

• **Menu Bar** — A series of menu buttons. See *Creating and Handling Menus on page 173* for more information.

• **Message Box** — A popup message box containing a text message, which can be blocking or nonblocking. See *Creating Popup Messages on page 190* for more information.

• **Option Menu** — A menu button that reflects the currently selected menu item. See *Creating and Handling Menus on page 173* for more information.

• **Popup Menu** — A menu that appears when the user presses the right mouse button over a parent widget. See *Creating and Handling Menus on page 173* for more information.

• **Radio Box** — A specified number of rows or columns of labeled toggle buttons. See *Creating a Radio Box on page 181* for more information.

• **Table Widget** — An editable 2D array of cells similar to a spreadsheet. See *Creating a Table Widget on page 200* for more information.

• **Text Area** — A static text label, a text entry field, or a full window editor. See *Creating a Text Widget on page 186* for more information.

• **Tool Box** — An array of graphic buttons (icons) arranged in a specified number of columns or rows. See *Creating a Button Box and a Tool Box on page 179* for more information.
Arranging Widgets in a Layout

A layout widget is like the canvas on which other widgets are drawn in whatever arrangement you specify. All types of widgets, except the root window and popup widgets, must be related to a layout widget.

The WwInit function creates a main window and one layout widget by default. Additional layout widgets can be created with the WwLayout function.

A layout widget allows three types of basic arrangements:

- Row/column layout
- Bulletin Board layout
- Form layout

In addition, layout widgets can be embedded within other layout widgets to create more complex GUIs.

Row/Column Layout

Row/column is the default layout. A row/column layout consists of widgets arranged either horizontally or vertically depending on the keywords used. The following commands create a row/column layout, where the widgets are horizontally aligned, with five pixels of space between each widget and widget borders three pixels wide. The result is shown in Figure 5-7.

```plaintext
top=WwInit('ww_ex3', 'Examples', layout, $ /Horizontal, Spacing=5, Border=3)
btn1=WwButtonBox(layout,'Button 1', 'CB')
btn2=WwButtonBox(layout,'Button 2', 'CB')
btn3=WwButtonBox(layout,'Button 3', 'CB')
status=WwSetValue(top, /Display)
WwLoop
```

![Figure 5-7](image) Horizontally aligned layout with three buttons (Motif style).
**Bulletin Board Layout**

To create a bulletin board layout, the WwInit or WwLayout function is used with the *Board* keyword.

Each widget is positioned on the bulletin board with the *Position* keyword, which specifies \( x \) and \( y \) coordinates. By default, widgets are placed in the upper-left corner of the bulletin board (coordinates \( x=0, y=0 \)).

For example, the following calls position button widgets on the bulletin board called *bboard*. Note that the positions of the buttons are specified with the *Position* keyword. The result is shown in *Figure 5-8*.

```wonderware
top=WwInit('ww_ex4', 'Examples', bboard, /Board)
btn1=WwButtonBox(bboard,'Button 1', 'CB', Position=[0,0])
btn2=WwButtonBox(bboard,'Button 2', 'CB', Position=[0,50])
btn3=WwButtonBox(bboard,'Button 3', 'CB', Position=[0,100])
status=WwSetValue(top, /Display)
WwLoop
```

*Figure 5-8* Bulletin board layout (Motif style).

**Form Layout: Attachments**

On a form layout, widgets are “attached” to one another. These attachments are specified with the keywords *Top*, *Bottom*, *Right*, and *Left*. You can specify widget attachments in relation to the parent widget or in relation to other child widgets. Many combinations of attachments are possible, and it is best to experiment with the attachment keywords to produce the desired effect.

For example, the following call creates a form layout called *ctrls*:

```wonderware
ctrls=WwLayout(top, /Form)
```
If no attachment keyword is specified, a child widget `bbox1` is placed in the upper left corner of the layout. For example:

```python
bbox1=WwButtonBox(ctrls, 'Click Here', 'buttonCB')
```

The result is illustrated in the following figure:

![Image of bbox1 placement](image)

The next call places another button in the layout. This attachment places the top of `bbox2` on the bottom of `bbox1`, as shown in the following figure.

```python
bbox2=WwButtonBox(ctrls, 'Click Here', 'buttonCB', Top=bbox1)
```

![Image of bbox1 and bbox2 placement](image)

The third button is attached to the right edge of the first button, `bbox1`, as shown in the following figure.

```python
bbox3=WwButtonBox(ctrls, 'Click Here', 'buttonCB', Left=bbox1)
```

![Image of bbox1, bbox2, and bbox3 placement](image)

Finally, a fourth button is attached to bottom of the third button and to the left of the second button, as shown in the following figure.

```python
bbox4=WwButtonBox(ctrls, 'Click Here', 'buttonCB', Left=Box2, Top=bbox3)
```

![Image of bbox1, bbox2, bbox3, and bbox4 placement](image)
Figure 5-9 summarizes the general effect of widget attachments specified for the widget Child_2 in relation to the widget Child_1.

You can also specify attachment keywords with a value of one (Keyword = 1). This positions widgets in relation to the parent layout. For example:

bbox1=WwButtonBox(layout, ‘Click Here’, ‘buttonCB’, /Bottom)

This call attaches the widget bbox1 to the bottom of the parent layout widget layout. These default attachments are always specified in relation to the parent widget.

Figure 5-10 summarizes the effect of some basic attachment keyword defaults (Keyword = 1).
Creating and Handling Menus

This section discusses the three basic types of WAVE Widgets menus:

• Menu bar
• Popup menu
• Option menu

Menu Bar

![Menu Bar Diagram]

*Figure 5-11* A menu bar. On the left no menu items are selected. On the right, the Fonts menu is selected, and an additional pullright menus is displayed.
A menu bar is a set of buttons that activate menus. When you select a menu button with the mouse, a pulldown menu appears with a set of menu items. A menu item can itself activate another menu, called a pullright menu, as shown in Figure 5-11.

The WwMenuBar function is used to create a menu bar. This function takes two parameters:

\[
\text{menubar} = \text{WwMenu}bar(parent, items)
\]

**NOTE** WwMenuBar can only occur once per layout widget on Windows.

The returned value, `menubar`, is the ID of the newly created menu bar widget. The `parent` parameter is the widget ID of the parent widget, often the ID of the layout widget. The `items` parameter is an unnamed structure containing all of the menu information. For detailed information on the `items` parameter, see *Defining Menu Items with Unnamed Structures* on page 176.

**Popup Menu**

![Example 5](example5.png)

**Figure 5-12** Simple popup menu, activated over a drawing area parent widget (Motif style).
A popup menu, shown in Figure 5-12, is a menu that appears when you press the MENU mouse button (usually the right button) while the cursor is positioned inside the menu’s parent widget.

The WwPopupMenu function is used to create a popup menu. This function takes two parameters:

\[
\text{menubar} = \text{WwPopupMenu}(\text{parent}, \text{items})
\]

The \textit{parent} parameter is the widget ID of the parent widget. The \textit{items} parameter is an unnamed structure containing all of the menu information. For detailed information on the \textit{items} parameter, see \textit{Defining Menu Items with Unnamed Structures} on page 176.

**Option Menu**

### Figure 5-13

On the left are three option menus, Fonts, Types, and Sizes. Each one displays the current selection. On the right, the Fonts menu is selected.

An option menu, shown in Figure 5-13, is a menu button that displays the current selection. When the user presses the appropriate mouse button (usually the left button for Motif) over an option menu button, a menu appears. After the user makes a selection, the option button’s text changes to reflect the current selection.

The WwOptionMenu function is used to create an option menu. This function takes three parameters:

\[
\text{options} = \text{WwOptionMenu}(\text{parent}, \text{label}, \text{items})
\]

The \textit{parent} parameter is the widget ID of the parent widget. The \textit{label} parameter is a text string containing a label for the menu. The \textit{items} parameter is an unnamed structure containing all of the menu information. For details on the \textit{items} parameter, see \textit{Defining Menu Items with Unnamed Structures} on page 176.
Menu Callbacks

When an active menu item is selected, the menu callback is called with the menu’s widget ID as the first parameter and the menu item index (1...n) as the second parameter.

For example, here is a very simple callback routine called MenuCB, which prints a message depending on which menu button is selected.

```
PRO MenuCB, wid, which
CASE which OF
  1: PRINT,'First Button Selected'
  2: PRINT,'Second Button Selected'
  3: PRINT,'Third Button Selected'
ENDCASE
END
```

Defining Menu Items with Unnamed Structures

Each of the menus discussed above takes a parameter (items) that contains the menu information. This parameter is defined as an unnamed structure.

An unnamed structure has the following general definition:

```
x = {, tag_name1 : tag_def1, tag_name2 : tag_def2,...}
```

For detailed information on unnamed structures, see the section the section Creating Unnamed Structures in Chapter 6 of the PV-WAVE Programmer’s Guide.

The following tag names and tag definitions can be used in the unnamed structure used to define menu items. For an example, see the next section.

- **title: 'name'** — (optional) Specifies a title for the menu.
- **callback: 'clbkname'** — The name of the callback routine to be executed when an active menu button is selected. Always place the callback name before the active button’s definition. The active items, which are described below, include button, icon, toggle, and menubutton.
- **button: 'labelname'** — The name of a pushbutton on the menu. A pushbutton is a button that calls a callback when selected.
- **icon: 'bitmap_filename'** — Creates an iconic (graphic) pushbutton. The bitmap filename is the full name of a file containing the icon’s bitmap.
- **toggle: 'labelname'** — Creates a “toggle” type of button. A toggle button contains a small box to the left of the label. When the button is selected, the box is highlighted. When the button is not selected, the box is not highlighted.
• menubutton: 'labelname' — The name of a menu button on the menu bar, or in the menu for pullright menus.

• menu: structure — An unnamed structure that defines the contents of menus. For a simple pulldown menu, the structure has only one level. Include pullright menus by embedding additional structures in the top-level structure.

• separator: value — Separates the previous from the next menu item. Possible values are:

<table>
<thead>
<tr>
<th>Values for Motif</th>
<th>Separator</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No line</td>
</tr>
<tr>
<td>1</td>
<td>Single line</td>
</tr>
<tr>
<td>2</td>
<td>Double line</td>
</tr>
</tbody>
</table>

• current: index (1 – n) — Used only for the option menu. Specifies the item to be selected as current when the option menu is created. This tag must be the last one in the list of tags in the structure definition.

Modifying Menu Items

WwMenuItem lets you dynamically update menus that have already been created. All menu items are placed in a parent menu pane, and the widget ID of the appropriate menu pane can be acquired using the Menus keyword of the WwMenuBar, WwPopupMenu, or WwOptionMenu function. For details on the WwMenuItem function, see WwMenuItem Function on page 298.

Example

The following example shows the code used to create the items for the menus shown in Figure 5-14. The menu bar contains three menus: Fonts, Size, and Icons. The contents of these menus are defined by the unnamed structure, menus, which is then passed to the WwMenuBar function. Note that the first menu, Fonts, contains a pullright menu called Adobe. This is defined as a separate unnamed structure embedded in the top-level structure.
Figure 5-14 Menu bar containing three menus. Menus are displayed when the user presses the SELECT mouse button (usually the left button) over the menu name. The menus for each button are shown below the menu bar.

top=WwInit('ww_ex5', 'Examples', layout)
menus =[{callback:'MenuCB',$
  menubutton:'Fonts',$
  ; Create menu button "Fonts" on the menu bar.
  menu:{,callback:'MenuCB',$
    menubutton:'Adobe',$
    ; A new menu is created by embedding another unnamed structure
    ; in the top-level structure. The menubutton tag creates a pullright
    ; menu called "Adobe". The pullright menu contains the toggle
    ; buttons "Normal", "Bold", and "Italic".
    menu:{,callback:'MenuCB',$
      toggle:'Normal',$
      toggle:'Bold',$
      toggle:'Italic'},$
    button:'Helvetica',$
    button:'Courier'},$
  button:'Helvetica',$
  button:'Courier'},$
          ; Create two more pushbuttons on the Fonts menu.
  menubutton:'Size',$
          ; Create a second menu button "Size" on the menu bar.
  menu:{,callback:'MenuCB',$
    button:'8',$
    button:'10',$
    button:'12'}},$
; Create three pushbuttons on the “Size” menu.
menubutton:‘Icons’,$
; Create a third menu button “Icons” on the menu bar.
menu:{,callback:‘MenuCB’,$
    pushpin:0,$
; Do not display a pushpin.
    title:‘Help’,$
; Create a title for the Icons menu.
    icon:getenv('WAVE_DIR')+ $
        '/xres/wxbm_btn_help_search’,$
    icon:getenv('WAVE_DIR')+ $
        '/xres/wxbm_btn_help_toc’,$
    icon:getenv('WAVE_DIR')+ $
        '/xres/wxbm_btn_help_topics’,$
    ; Place four icon buttons on the Icons menu. Full pathnames to 
    ; bitmap images are given to produce the icon pictures.
    separator:1,$
; Insert a single-line separator.
    icon:getenv('WAVE_DIR')+ $
        '/xres/wxbm_btn_help_quit’}
; Place the last icon below the line.
bar = WwMenuBar(layout, menus)
; The call to WwMenuBar creates the menu bar. The ‘layout’ 
; parameter is the widget ID of the parent layout widget. For 
; example: layout = WwLayout(top, /Horizontal, /Spacing=5).
; The value of the ‘menus’ parameter is the multi-level unnamed 
; structure defined above.

status=WwSetValue(top, /Display)
WwLoop

---

**Creating a Button Box and a Tool Box**

A button box and a tool box serve similar functions — both trigger specific actions when the user clicks on an item. A button box contains an array of labeled buttons, organized in rows or columns. A tool box contains an array of icons (graphical buttons), also arranged in rows or columns.

Buttons and icons are usually used to apply changes, confirm decisions, display new windows, or start new applications. When the user clicks on a button or icon, its three-dimensional appearance inverts, so that it looks like the button has been depressed. When the button or icon is released, it returns to its normal appearance.
**Button Box Example**

![Button Box Example](image)

**Figure 5-15** A button box containing five buttons.

A button box is created with the WwButtonBox function. In the following example layout is the parent widget. ButtonCB is the name of the procedure that is executed when a button is selected. The buttons are arranged horizontally and spaced 20 pixels apart. WwButtonBox is passed two parameters — the button box widget ID and a value that corresponds to the selected button. The result is shown in *Figure 5-15*.

```plaintext
top=WwInit('ww_ex6', 'Examples', layout)
labels = ['Quit','Dialog','Message', 'FileSelection', 'Command']
bbox=WwButtonBox(layout, labels, 'ButtonCB', $ /Horizontal, Spacing=20)
status=WwSetValue(top, /Display)
WwLoop
```

**Tool Box Example**

![Tool Box Example](image)

**Figure 5-16** A toolbox containing four icons.

A tool box is created with the WwToolBox function. This example creates a tool box containing four icons, arranged in two columns, which are specified in the string array pixmaps. DrawnCB is the name of the callback routine (not shown) that is executed when a button is selected. The result is shown in *Figure 5-16*. 
Creating a Radio Box

Figure 5-17 A radio box (Motif style).

A radio box contains a set of buttons, usually used to set or display the state of a variable, process, or action. Radio box buttons can be either exclusive or nonexclusive. If the buttons are exclusive, only one can be selected at a time (like the buttons on a car radio). If the buttons are set to nonexclusive, any number of them can be selected at once. When a button is selected, its appearance changes to reflect this. Radio box buttons can also be labeled.
**Example**

A radio box is created with the WwRadioBox function. In this example, the keyword `Nofmany` specifies that the buttons are to be nonexclusive. The `Border` keyword specifies the thickness, in pixels, of the border around the buttons. The `Spacing` keyword specifies the space, in pixels, between buttons. `RadioCB` is the name of a callback routine that is executed when a button is selected. The result is shown in Figure 5-17.

```plaintext
top=WwInit('ww_ex8', 'Examples', layout)
labels=['System', 'Owner', 'Group'] ; Create three labels for the radio buttons.
rbox=WwRadioBox(layout, labels, 'RadioCB', $ /Vertical, Border=2, Spacing=20, /Nofmany)
; Create the radio button box.
status=WwSetValue(top, /Display)
WwLoop
```

---

**Creating a Controls Box with Sliders**

![Example3](image)

Figure 5-18 A controls box with three sliders (Motif style).

A controls box is a box containing “sliders”. A slider can be used to change numerical values interactively by positioning the pointer on the slider, pressing the left mouse button, and dragging. A controls box can have any number of sliders oriented either horizontally or vertically in a specified number of rows or columns. In addition, a slider can have an input text field in which the user can enter an exact value.
A controls box is created with the WwControlsBox function.

**Example**

In this example, a controls box containing three sliders, labeled *Pressure*, *RPM*, and *Temperature*, is created. They are arranged vertically. The keyword *Text* creates a text input field for each slider. Whenever a slider is moved, the callback routine named *SliderCB* is executed. The result is shown in Figure 5-18.

```wax
top=WwInit(‘ww_ex9’, ‘Examples’, layout)
labels=[‘Pressure’,‘RPM’,‘Temperature’]  
   ;Create the slider labels.
ranges=[0,100,2000,4000,50,150]  
   ;Specify the ranges of each slider.
controls = WwControlsBox(layout, labels, $  
   ranges, ‘SliderCB’, /Vertical, /Text, $  
   Foreground=‘red’, Background=‘yellow’, Basecolor=‘blue’)  
   ;Create the controls box.
status=WwSetValue(top, /Display)
WwLoop
```

---

**Creating a Drawing Area**

![Figure 5-19](image)  

*Figure 5-19* A drawing area (Motif style).
A drawing area is a window in which an application can display plots or images. Both horizontal and vertical scroll bars are attached to the drawing area. The scroll bars can be used to pan across a drawing that is too large to fit in the window. A drawing area is created with the WwDrawing function.

Whenever a drawing area widget is created, WwDrawing automatically associates the drawing area with a PV-WAVE window index (see the WINDOW command in the PV-WAVE Reference for information on the window index). Then, the drawing area callback is executed and the graphics are displayed. The callback is also executed on systems that do not provide backing store for when the drawing window is obscured and then redisplayed.

**Example 1: Basic Drawing Area**

In this example, a drawing area window is created. The parent widget is called layout, the window index of the PV-WAVE window used is 1. Note that the size of the drawing area is 256 x 256, while the total size of the drawing is 512 x 512 (the image is larger than the actual drawing area). Scroll bars are provided for moving the image inside the drawing area. The result is shown in Figure 5-19.

To run this example, enter the callback procedure in a file and compile it with the .RUN command. Then enter the widget commands at the WAVE> prompt.

**Callback Procedure**

```plaintext
PRO DrawCB, wid, data
    COMMON draw, img
    PRINT, 'Draw'
    TV, img
END
```

**Widget Commands**

```plaintext
top=WwInit('ww_ex10', 'Examples', layout)
COMMON draw, img
LOADCT, 5, /Silent
img=BYTARR(512,512)
Openr,1, !Data_Dir + 'head.img'
READU,1,img
CLOSE, 1
draw=WwDrawing(layout, 1, 'DrawCB', [256,256], [512,512])
status=WwSetValue(top, /Display)
WwLoop
```
Example 2: User Resizes the Drawing Area

This example demonstrates how the drawing area can be dynamically resized when the user resizes the main window.

To run this example, enter the code into a procedure file, and execute it with the .RUN command.

```
PRO DrawCB, wid, index
   COMMON Trdraw, draw, area, widx
   print, 'Draw', !D.x_vsize, !D.y_vsize, index, widx
   x = indgen(100)
   plot, x
END

PRO Testresize
   COMMON Trdraw, draw, area, widx
   ; Initialize toolkit, create form layout
   top = WwInit('testresize','Test',layout, $  
      Background='Green', /Form)
   Create drawing area
   widx = -1
   draw = WwDrawing(layout, widx, 'DrawCB', $  
      [256, 256], [512, 512], /Noscroll, $  
      Area = area, /Right, /Left, /Top, /Bottom)
   status = WwSetValue(top, /Display)
   WtLoop
END
```

Example 3: Resizing the Drawing Area Programmatically

This example demonstrates how to resize the drawing area programmatically using WwSetValue.

To run this example, enter the code into a procedure file, and execute it with the .RUN command.

```
PRO ButtonCB, wid, which
   COMMON Test, draw, top, size
   CASE which OF
      1: BEGIN
```

185
size = 600
status = WwSetValue(draw, [600, 600])
END
2: BEGIN
  erase, 0
tvsc1, dist(size)
END
ENDCASE
END

; Draw the image.
PRO DrawingCB, wid, which
  COMMON Test, draw, top, size
tvsc1, dist(size)
END
PRO Testdraw
  COMMON Test, draw, top, size
  size = 400
  top = WwInit('testdraw', 'Testdraw', layout, $
    /Vertical, Title = 'Test Drawing', Position = position)
    ; Create Pushbutton.
b = ['Resize','Redraw']
pushb = WwButtonBox(layout, b, 'ButtonCB', $
  /Vertical, Border = 0, Spacing = 10, Position = [10, 400])
    ; Create Drawing.
drawing_surf = WwDrawing(layout, 1, $
  'DrawingCB', [400, 400], [400, 400], $
  Area = draw, Position = [0, 0])
status = WwSetValue(top, /Display)
WwLoop
    ; Waiting for callbacks.
END

Creating a Text Widget

The WwText function can be used to create three kinds of text widgets:

- A single-line read-only label.
- A single-line editable text field, used for user input.
• A multi-line text window that can be read-only or editable. Horizontal and vertical scroll bars allow the user to scroll through the multi-line text window.

**Single-line Label (Read-only)**

To create a single-line read-only text label, use the WwText function with the `Label` keyword. For an example, see *Static Label and Editable Text Field Example* below.

**Single-line Editable Text Field**

![Editble Text Field with Label](image)

Figure 5-20 An editable text field with a label.

To create a single-line editable text field, use the WwText function. This text widget is used primarily for applications that require the user to enter a value or a string. WwText returns a string, which can be passed to the callback for processing.

**Static Label and Editable Text Field Example**

This example creates an editable text field with a label. The left edge of the text field widget is attached to the right edge of the label widget. The result is shown in *Figure 5-20*.

```wavedata
top=WwInit('ww_ex11', 'Examples', layout, /Form)
    ; Initialize WAVE Widgets and create the form layout widget.
label=WwText(layout, /Label, Text='This is Label')
    ; Create the label widget.
text=WwText(layout, 'TextCB', Cols=40, left=label)
    ; Create the single-line text field widget, attaching it to the right edge of the label widget.
status=WwSetValue(top, /Display)
WwLoop
```
Multi-line Text Window

To create a multi-line text window, use the WwText function with the Col and Rows keywords to specify the height and width of the text area. If you use the Read keyword, the text area is read-only. If this keyword is not used, then the user can edit the text using the standard editing keys on the keyboard or the mouse (for example cut and paste).

Example

This example creates a multi-line, read-only text window 40 columns by 20 rows, and displays the text from the file Tips. The result is shown in Figure 5-21.

```
top=WwInit('ww_ex12', 'Examples', layout)
filename = getenv('WAVE_DIR')+ '/Tips'
text=WwText(layout, 'TextCB', /Read, $ File=filename, Cols=40, Rows=20)
status=WwSetValue(top, /Display)
WwLoop
```

Figure 5-21  A multi-line text window (Motif style).
Creating a Scrolling List

A scrolling list allows the user to select one or more items from a list of items. The “items” in a scrolling list are text strings, defined in a string array. Scroll bars are provided for lists that are too long to display in the scrolling list window.

Use the WwList function to create a scrolling list.

**Selection Mode**

You can create a scrolling list in one of two selection modes: single or multiple selection.

**Single Selection Mode**

This is the default selection mode. Single selection means that the user can select one item at a time. To select an item, the user positions the pointer over the item and clicks the SELECT mouse button, usually the left button. If the user selects another item, the first item is deselected. This then executes the `selectCallback` routine.

If the user double-clicks on an item, the `defaultCallback` routine is executed.

**Multiple Selection Mode**

In multiple selection mode, the user can select more than one item from the list. The first item is selected with left mouse button. Additional items can then be selected using left mouse button. To deselect an item, the user clicks the left mouse button on it.
Scrolling List Callbacks

Two callbacks are used with the WwList function. The first one, `selectCallback`, is called whenever the user selects an item (single-clicks on it). The second callback, `defaultCallback`, is called when the user double-clicks on an item. This callback is called “default” because it usually executes a default action. For example, the Motif file selection widget operates in this manner. When the user clicks on a file name, the text is placed in a text input field. When the user double-clicks on a file name, the file is selected, the selection widget is dismissed, and the `defaultCallback` is called. For an illustration of the Motif file selection widget, see Creating a File Selection Widget on page 196.

Example

The following example creates a scrolling list containing the names of holidays. The `Visible` keyword specifies the number of items that are displayed in the scrolling list at a time. The `Multi` keyword sets the scrolling list to the multiple selection mode. The result is shown in Figure 5-22.

```plaintext
top=WwInit('ww_ex13', 'Examples', layout)
items = ['PresidentsDay', 'St.PatricksDay',$
'    'Easter', 'MemorialDay', '4th of July',$
'    'LaborDay', 'Halloween', 'Thanksgiving',$
'    'Hanukkah', 'Christmas', 'New Years Eve']
    ; Define the list of items.
list=WwList(layout, items, 'ListCB',$
    'DefaultCB', Visible=7,/Multi, Left=rbox, Top=controls)
    ; Create a scrolling list widget that displays the list of holidays.
status=WwSetValue(top, /Display)
WwLoop
```

Creating Popup Messages

![Image of Message Window]

A message, for your information.

OK  Cancel  Help
A popup message is a window that contains some text. Usually it informs the user of a condition — a warning or the confirmation of a choice, for instance — then is dismissed when the user clicks on the **OK** button. No other interaction is required.

The **WwMessage** function is used to create message windows.

### NOTE
Message windows are *popup widgets*. This means that they must have an intermediate widget, such as a button, as their parent. The popup widget appears after the user selects the intermediate button. See [Message Box Example](#) on page 192 for more information.

---

### Blocking vs. Nonblocking Windows

A message window can be blocking or nonblocking. Blocking means that no other user action can occur until the message or dialog is confirmed — user clicks on **OK** — or the message box is dismissed. The blocking window “blocks” the user from performing other actions as long as the window remains on the screen.

A nonblocking message window can remain on the screen while the user performs other actions. The nonblocking dialog or message does not “block” the user from performing other actions.

The **WwMessage** function has a **Block** and a **Nonblock** keyword. Use the **Block** keyword to create a blocking window and the **Nonblock** keyword to create a nonblocking window.

### Types of Message Windows

You can create four types of message boxes. These types are available only with the Motif message widget. They are shown in *Figure 5-24*.

- Information message — Specified with the **Info** keyword.
- Working message — Specified with the **Working** keyword.
- Warning message — Specified with the **Warning** keyword.
- Question message — Specified with the **Question** keyword.
Figure 5-24 The four types of message windows. Clockwise from upper-left: Information, Working, Question, and Warning.

**Default Message Box Buttons**

By default, three buttons appear along the bottom edge of message windows. They are used to confirm or cancel the message.

**Motif Buttons**

OK — Confirms the message.

Cancel — Cancels the message.

Help — Not supported.

**Message Box Example**

This example creates a row of buttons you can click on to display the four different types of Motif message boxes. To run the example, enter the callback procedures in a file and compile them with the .RUN command. Then enter the widget commands at the WAVE> prompt. The result is shown in Figure 5-23.

**Callback Routines**

```
PRO MessageOK, wid, data
    print,'Message OK'
END

PRO MessageCancel, wid, data
    print,'Message Cancel'
```
PRO MbuttonCB, wid, data

case data of
1: message=WwMessage(wid, "$ 'This is a Test Message','MessageOK', "$ 'MessageCancel',TITLE='Information')
2: message=WwMessage(wid, "$ 'This is a Test Message','MessageOK', "$ 'MessageCancel', /Working, $ TITLE='Working')
3: message=WwMessage(wid, "$ 'This is a Test Message','MessageOK', "$ 'MessageCancel', /Warning, TITLE='Warning')
4: message=WwMessage(wid, "$ 'This is a Test Message','MessageOK', "$ 'MessageCancel', /Question, TITLE='Question')
ENDCASE

END

Widget Commands

top=WwInit('ww_ex14', 'Examples', layout)

button=WwButtonBox(layout, ['Information', "$ 'Working', 'Warning', 'Question'], 'MbuttonCB')

status=WwSetValue(top, /Display)

WwLoop

Creating Dialog Boxes
Figure 5-25  A dialog box (Motif-style).

A dialog box requires user interaction. For instance, the users may enter some text in the dialog box, then “accept” the entry by clicking on a button. If the users do not wish to apply the change to the dialog box, they can click on another button to dismiss it.

The WwDialog function creates dialog boxes.

A Dialog is a Popup Widget

Dialog boxes are *popup widgets*. This means that they must have an intermediate widget, such as a button, as their parent. The popup widget appears after the user selects this intermediate button. See *Dialog Box Example on page 195* for more information.

Blocking vs. Nonblocking Windows

A dialog window can be blocking or nonblocking. Blocking means that no other user action can occur until the dialog is confirmed — the user clicks on OK or Confirm — or until the dialog box is dismissed. The blocking window “blocks” the user from performing other actions as long as the window remains on the screen.

A nonblocking dialog can remain on the screen while the user performs other actions. The nonblocking dialog does not “block” the user from performing other actions.

The WwDialog function has a Block and a Nonblock keyword. Use the Block keyword to create a blocking window and the Nonblock keyword to create a nonblocking window.

Default Dialog Box Buttons

By default, three buttons appear along the bottom edge of dialog boxes. These buttons are used to confirm or cancel the dialog box.

*Motif Buttons*

OK — Confirms the input in the dialog.

Cancel — Cancels the dialog.

Help — Not supported.
Dialog Box Example

The following example creates a button that you can click on to display a dialog box. To run the example, enter the callback procedures in a file and compile them with the .RUN command. Then enter the widget commands at the WAVE> prompt.

The “Type something” string is a label for the text input field. The DialogOK parameter is the name of a callback that is executed when the user clicks on the OK button. The DialogCancel parameter is the name of a callback that is executed when the Cancel button is selected. The result is shown in Figure 5-25.

Callback Procedures

PRO DbuttonCB, wid, data
    select=WwDialog(wid,‘Type something:’,$’
           ‘DialogOK’, ‘DialogCancel’, Title=‘Type’)
END

PRO DialogOK, wid, text
    PRINT,‘Dialog OK’
    value = WwGetValue(text)
    PRINT, value
END

PRO DialogCancel, wid, data
    PRINT,’Dialog Cancel’
END

Widget Commands

top=WwInit(‘ww_ex15’, ‘Examples’, layout)
button=WwButtonBox(layout, ‘Dialog Box’, ’DbuttonCB’)
status=WwSetValue(top, /Display)
WwLoop
Creating a File Selection Widget

A file selection widget lets the user move through directories and select files. File selection widgets are created with the WwFileSelection function.

A File Selection Widget is a Popup Widget

File selection widgets are *popup widgets*. This means that they must have an intermediate widget, such as a button, as their parent. The popup widget appears after the user selects this intermediate button. See *File Selection Example* on page 197 for more information.

Blocking vs. Nonblocking Windows

A file selection window can be blocking or nonblocking. Blocking means that no other user action can occur until the message or dialog is confirmed (user clicks on...
OK, Apply, Cancel, or some other confirming button). The blocking window “blocks” the user from performing other actions as long as the window remains on the screen.

A nonblocking file selection window can remain on the screen while the user performs other actions. The nonblocking window does not “block” the user from performing other actions.

The WwFileSelection function has a Block and a Nonblock keyword. Use the Block keyword to create a blocking window and the Nonblock keyword to create a non-blocking window.

File Tool Contents

The file tool lets the user move through directories, view their contents, and select files. It consists of:

- A text field where the user enters a directory name to display subdirectories and files.
- A list of directories and files.
- A text input field for displaying or editing a filename.
- The buttons: OK, Filter, Cancel, and Help.

See Figure 5-26 for an example Motif file selection box.

File Selection Example

The following example creates a button that you can click on to display a file selection widget. To run the example, enter the callback procedures in a file and compile them with the .RUN command. Then enter the widget commands at the WAVE> prompt.

FileOK and FbuttonCB are names of callback routines. The Title keyword specifies a name for the search tool. The result is shown in Figure 5-26.

Callback Procedures

```
PRO FbuttonCB, wid, data
    file = WwFileSelection(wid,’FileOK’,’FileCancel’, Title=’Search’)
END

PRO FileOK, wid, shell
    value = WwGetValue(wid)
```
PRINT, value
status = WwSetValue(shell, /Close)
END

**Widget Commands**

top=WWInit('ww_ex16', 'Examples', layout)
button=WwButtonBox(layout, 'File Tool', 'FbuttonCB')
status=WwSetValue(top, /Display)
WwLoop

---

**Creating a Command Widget**

![Command Widget Diagram](image)

Figure 5-27 A command widget (Motif style).

A command widget is used for command entry and provides a built-in command history mechanism. The command widget includes a text input field, a label for the text input field, and a command history window.

The WwCommand function is used to create a command widget. Use WwGetValue in the callback routine to obtain the text strings entered from the command widget.

The user types a command in the text input field and presses <Return> to execute the command. The command is then added to the end of the command history win-
When the user clicks on a command in the command history window, the
command is displayed in the text entry field, ready to be executed. The user can
double-click on a command in the command history list to execute it directly. This
also adds the command to the end of the list.

NOTE Command widgets are *popup widgets*. This means that they must have an
intermediate widget, such as a button, as their parent. The popup widget appears
after the user selects this intermediate button. See the following Example section
for more information.

Example

The following example shows a simple WwCommand call. To run the example,
enter the callback procedures in a file and compile them with the .RUN command.
Then enter the widget commands at the WAVE> prompt.

CommandOK is a callback that is executed when the user enters the command and
presses <Return>, or double-clicks the command from the history list. Command-
Done is a callback that is executed when the user quits the command window. The
Title keyword specifies a name for the command window, and Position specifies its
location on the screen. The result is shown in *Figure 5-27*.

**Callback Procedures**

```plaintext
PROC buttonCB, wid, data
  command = WwCommand(wid, 'CommandOK', 'CommandDone', Title= $
    'Command Entry Window', Position=[300,300])
END

PROC CommandOK, wid, shell
  value = WwGetValue(wid)
    ; Obtain the string entered in the text input field.
  PRINT, value
END

PROC CommandDone, wid, shell
  status = WwSetValue(shell, /Close)
END
```

**Widget Commands**

```plaintext
  top=WwInit(‘ww_ex17’, ‘Examples’, layout)
  button=WwButtonBox(layout, ‘Command’, ‘CbuttonCB’)
  status=WwSetValue(top, /Display)
  WwLoop
```
Creating a Table Widget

![Table Widget Image]

**Figure 5-28** A table widget.

A table widget is used for displaying and editing a 2D array of cells. The example table shown in **Figure 5-28** contains scroll bars which can be used to display cells that are currently hidden.

The WwTable function is used to create a table widget. A variety of selection and editing methods are available within a table created with WwTable. For detailed information on the many keywords and options of WwTable and the code used to produce **Figure 5-28**, see *WwTable Function* on page 339.
Setting Colors and Fonts

Colors and fonts are common attributes of all WAVE Widgets. Keywords are provided for setting these attributes.

Setting Colors

Many WAVE Widgets provide keywords for setting colors. The keywords include: Basecolor, Background, and Foreground.

Basecolor — The color of the “container” or “box” for the following widgets: ButtonBox, RadioBox, ToolBox, and ControlsBox.

Background — The color of a button.

Foreground — The color of the text on a button.

Figure 5-29 illustrates the use of each color keyword.

![Figure 5-29 How the color keywords are applied.](image)

The color keywords have the form:

```
Keyword = 'colorname'
```

where colorname is the name of an X library color. For example:

```
Background = 'Skyblue'
```

The names of colors can be found in these files on most UNIX and most OpenVMS systems:

(UNIX) /usr/lib/X11/rgb.txt

(OpenVMS) SYS$MANAGER:DECW$RGB.COM

On Windows, the file containing the color names is:

(Windows) <maindir>\wave\lib\std\windows\rgb.txt

where <maindir> is the main PV-WAVE directory.
NOTE For Windows, this file is a reference file only. Any modifications made to this file will not affect the colors used by PV-WAVE.

If you cannot find these color files on your system, see your System Administrator.

**Predefined Colors for Windows Systems**

In addition to the colors listed in the rgb.txt file, you may use any of the following predefined Windows system colors as your widget colors.

<table>
<thead>
<tr>
<th>Windows System Colors</th>
<th>Corresponding Widget Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR_3DDKSHADOW</td>
<td>Dark shadow for 3D display elements.</td>
</tr>
<tr>
<td>COLOR_3DFACE,</td>
<td>Face color for 3D display elements.</td>
</tr>
<tr>
<td>COLOR_BTNFACE</td>
<td></td>
</tr>
<tr>
<td>COLOR_3DHILIGHT</td>
<td>Highlight color for 3D display elements (for edges facing the light source).</td>
</tr>
<tr>
<td>COLOR_BTNHIGHLIGHT,</td>
<td></td>
</tr>
<tr>
<td>COLOR_BTNHIGHLIGHT,</td>
<td></td>
</tr>
<tr>
<td>COLOR_3DLIGHT</td>
<td>Light color for 3D display elements (for edges facing the light source).</td>
</tr>
<tr>
<td>COLOR_3DSHADOW,</td>
<td>Shadow color for 3D display elements (for edges facing away from the light source).</td>
</tr>
<tr>
<td>COLOR_BTNSHADOW</td>
<td></td>
</tr>
<tr>
<td>COLOR_ACTIVEBORDER</td>
<td>Active window border.</td>
</tr>
<tr>
<td>COLOR_ACTIVECAPTION</td>
<td>Active window caption.</td>
</tr>
<tr>
<td>COLOR_APPWORKSPACE</td>
<td>Background color of multiple document interface (MDI) applications.</td>
</tr>
<tr>
<td>COLOR_BACKGROUND,</td>
<td>Desktop.</td>
</tr>
<tr>
<td>COLOR_DESKTOP</td>
<td></td>
</tr>
<tr>
<td>COLOR_BTNTEXT</td>
<td>Text on push buttons.</td>
</tr>
<tr>
<td>COLOR_CAPTIONTEXT</td>
<td>Text in caption, size box, and scroll bar arrow box.</td>
</tr>
<tr>
<td>COLOR_GRAYTEXT</td>
<td>Grayed (disabled) text. This color is set to 0 if the current display driver does not support a solid gray color.</td>
</tr>
<tr>
<td>COLOR_HIGHLIGHT</td>
<td>Items selected in a control.</td>
</tr>
</tbody>
</table>
Specifying XtDefaultForeground or XtDefaultBackground causes the widget to use the appropriate Windows system colors.

<table>
<thead>
<tr>
<th>Windows System Colors</th>
<th>Corresponding Widget Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR_HIGHLIGHTTEXT</td>
<td>Text of selected items in a control.</td>
</tr>
<tr>
<td>COLOR_INACTIVEBORDER</td>
<td>Inactive window border.</td>
</tr>
<tr>
<td>COLOR_INACTIVECAPTION</td>
<td>Inactive window caption.</td>
</tr>
<tr>
<td>COLOR_INFOBK</td>
<td>Color of text in an inactive caption.</td>
</tr>
<tr>
<td>COLOR_INFOTEXT</td>
<td>Text color for tool tip controls.</td>
</tr>
<tr>
<td>COLOR_MENU</td>
<td>Menu background.</td>
</tr>
<tr>
<td>COLOR_MENUTEXT</td>
<td>Text in menus.</td>
</tr>
<tr>
<td>COLOR_SCROLLBAR</td>
<td>Scroll bar gray area.</td>
</tr>
<tr>
<td>COLOR_WINDOW</td>
<td>Window background.</td>
</tr>
<tr>
<td>COLOR_WINDOWFRAME</td>
<td>Window frame.</td>
</tr>
<tr>
<td>COLOR_WINDOWTEXT</td>
<td>Text in windows.</td>
</tr>
</tbody>
</table>

Setting Fonts on UNIX and OpenVMS

The Font keyword is used to set the font used to create text in a widget. The Font keyword has the form:

\[ Font = 'fontname' \]

where fontname is the name of a font available on your system. On UNIX, for example, Font may be specified as follows:

\[ Font = '-b&h-lucida-bold-r-normal-sans-14-*' \]

**TIP** Use the command xlsfonts to obtain a list of fonts available on your X server.

If you inadvertently specify a font that cannot be found by the X server, a different font will be substituted.
Setting Fonts on Windows

On Windows, use the MSFont keyword to specify fonts for use in WAVE Widgets. The MSFont keyword is specified as a string of the following form:

'face_name, point size, attribute'

where face_name specifies the type face, and attribute specifies font attributes such as bold, italic, underline, etc.

For example: MSFont = ‘Arial, 8, bold’

You can also specify Windows system fonts as shown in the following table.

<table>
<thead>
<tr>
<th>Windows System Fonts</th>
<th>Corresponding Widget Fonts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI_FIXED</td>
<td>Windows fixed-pitch (monospace) system font.</td>
</tr>
<tr>
<td>ANSI_VAR</td>
<td>Windows variable-pitch (proportional space) system font.</td>
</tr>
<tr>
<td>DEFAULT_GUI</td>
<td>Windows 95/NT 4.0 only: Default font for user interface objects such as menus and dialog boxes.</td>
</tr>
<tr>
<td>DEVICE_DEFAULT</td>
<td>Windows NT only: Device-dependent font.</td>
</tr>
<tr>
<td>OEM_FIXED</td>
<td>Original equipment manufacturer (OEM) dependent fixed-pitch (monospace) font.</td>
</tr>
<tr>
<td>SYSTEM_VAR</td>
<td>By default, Windows uses the system font to draw menus, dialog box controls, and text. In Windows versions 3.0 and later, the system font is a proportionally spaced font; earlier versions of Windows used a monospace system font.</td>
</tr>
<tr>
<td>SYSTEM_FIXED</td>
<td>Fixed-pitch (monospace) system font used in Windows versions earlier than 3.0. This font is provided for backwards compatibility with earlier versions of Windows.</td>
</tr>
</tbody>
</table>

If the font you specify isn’t supported on your system, Windows substitutes another font, which is usually the system font.

Using a Resource File to Set Colors and Fonts

Characteristics such as foreground color, background color, and font are known as resources, and these characteristics can be initialized using a resource file. WAVE Widgets provides the ability to specify these resources with the keywords Foreground, Background, Basecolor, and Font, as discussed in the previous two sections.
When the Resource File is Checked

If widget resources are not specified with the keywords *Foreground*, *Background*, *Basecolor*, and *Font*, PV-WAVE will query the X resource database and use the values defined there.

NOTE Using a resource file for WAVE Widgets is optional.

Adding resources to the resource database is a two step process.

- The first step is to create a resource file. The following is a sample resource file for setting application-wide resources for foreground color, background color, and font of a WAVE Widgets application called *my_gui*:

  ```
  my_gui*foreground:Black
  my_gui*background: Cyan
  my_gui*font: fixed
  ```

  The application name *my_gui* is defined in the WwInit command. For example:

  ```
  top = WwInit('my_gui', 'Examples', layout, /Vertical)
  ```

  You can also put the resource names in your .Xdefaults file, if you only need to customize your own version of the software. Either way, the new values of the resources will not take effect until the next time PV-WAVE is started.

- The second step is to install the resource file into the X resource database. The simplest way to install resources is to merge your resource file with X resource database using the following command:

  ```
  xrdb -merge resource_filename
  ```

  where *resource_filename* is the name of your resource file.

NOTE It is also possible to set the resources for particular widgets or groups of widgets in your application. To do this, you will have to know the names of all or some of the widgets in the widget hierarchy. (A widget’s name is determined by its placement in the hierarchy.) The more specific the widget hierarchy you provide, the fewer the number of widgets that are affected by the change.

For more information about how widgets are related to one another, see *The Widget Hierarchy* on page 166.

To find out widget names of particular WAVE Widgets, refer to the appropriate WAVE Widgets .pro files located in the standard library directory:

(UNIX) $VNI_DIR/wave/lib/std/motif
Setting and Getting Widget Values

Most of the widgets have values associated with them that are set when the widget is created.

After a widget is created, these values can be changed or obtained using the WwGetValue or WwSetValue routines.

The values that are set or obtained with WwSetValue and WwGetValue differ from one WAVE Widgets routine to another. Refer to the specific routine descriptions in Chapter 7 for details.

For example, when WwSetValue is passed a widget ID from a widget created with WwList, the value that is set is an array of strings to replace the current items in the list. If WwGetValue is passed the widget ID from a widget created with WwList, it returns a string array containing the selected items in the scrolling list.

The following callback uses WwGetValue to obtain the list of selected items from a scrolling list. Then, WwSetValue is used to replace the selected items with new strings. To run this example, enter the callback routine in a file and compile it with the .RUN command. Then enter the widget commands at the WAVE> prompt.

Callback Procedure

PRO ListCB, wid, data
print, 'Item Selected'
value = WwGetValue(wid)
   ; Obtain the values of the selected items in the scrolling list.
print, value

print, 'Setting..'
status=WwSetValue(wid, ['First', 'Second', 'Third'])
   ; Set the value of the scrolling list widget whose ID is wid to
   ; the strings First, Second, and Third.
END
**Widget Commands**

```plaintext
top=WwInit('ww_ex18', 'Examples', layout)
items = ['Presidents Day', 'St.Patricks Day', 'Easter', 'Memorial Day', '4th of July',
    'Labor Day', 'Halloween', 'Thanksgiving', 'Hanukkah', 'Christmas', 'New Years Eve']
    ; Define the list of items.
list=WwList(layout, items, 'ListCB',
    'DefaultCB', Visible=7,/Multi, Left=rbox, Top=controls)
    ; Create a scrolling list widget that displays the list of holidays.
status=WwSetValue(top, /Display)
WwLoop
```

**Passing and Retrieving User Data**

All WAVE Widgets can carry the user-specified value of a variable. This allows the developer to store the copy of the variable with the widget in one routine and retrieve it in another routine. Any value can be stored and retrieved; it is up to the discretion of the programmer.

This feature is useful for passing values between routines without using Common Block variables.

To store the value 111 with a widget, use the following command:

```plaintext
status = WwSetValue(widgetID, Userdata=111)
```

To retrieve the value of Userdata from the widget, use the command:

```plaintext
value = WwGetValue(widgetID, /Userdata)
```

**Example**

The following example shows a practical use for passing a value with the Userdata keyword to close an application when the user clicks on a specified button. The value of the top-level widget, top, is passed from the widget creation procedure myap to the callback procedure ButtonCB via the Userdata keyword. This value is then used in the WwSetValue function to close the application by destroying the top-level widget when the user clicks on Quit. You can run this example by typing the callback procedures into a file and compiling them with the .RUN command. Then enter the application procedure in a file and run it.
Callback Procedure

PRO ButtonCB, wid, data
CASE data OF
  1: BEGIN
      top=WwGetValue(wid, /Userdata)
      ; Get the value of the top-level widget.
      PRINT, top
      status=WwSetValue(top, /Close)
      END
  2: PRINT, 'Dialog Selected'
  3: PRINT,'Message Selected'
ENDCASE
END

PRO RadioCB, wid, which
CASE which OF
  1: PRINT,'First Toggle Selected'
  2: PRINT,'Second Toggle Selected'
  3: PRINT,'Third Toggle Selected'
ENDCASE
value = WwGetValue(wid)
print, value
END

Application Procedure

PRO myap
top=WwInit('ww_ex19', 'Examples', layout,$
          /Vertical, Spacing=30, Border=10)
blabels = ['Quit','Dialog','Message']
bbox=WwButtonBox(layout, blabels, $
                  'ButtonCB', /Horizontal, Spacing=20)
status=WwSetValue(bbox, Userdata=top)
          ; Store the value of the top-level widget with the Userdata keyword.
rlabels=['System','Owner','Group']
rbbox=WwRadioBox(layout,rlabels, 'RadioCB', $
                   /Vertical, Border=2, Spacing=20, $
                   Top=controls)
status=WwSetValue(top, /Display)
WwLoop
END
Managing Widgets

Besides colors, fonts, and userdata values, two additional widget attributes can be managed by the developer: widget visibility and sensitivity.

Showing/Hiding Widgets

The *Show* and *Hide* keywords to the WwSetValue function control whether a widget is visible or not. By default, all widgets are shown when they are created.

To hide a widget, use the *Hide* keyword:

```
status = WwSetValue(widget, /Hide)
```

To show a hidden widget, use the *Show* keyword:

```
status = WwSetValue(widget, /Show)
```

In these functions, the *widget* parameter is the widget ID of the widget you want to show or hide.

You can also test to determine if a widget is shown or hidden, using the *Shown* keyword in the WwGetValue function.

```
shown = WwGetValue(widget, /Shown)
```

If the widget is shown, the function returns 1; if hidden, 0 is returned.

Widget Sensitivity

If a widget is sensitive, some action will occur when the user selects the widget. For example, if the user clicks on a button that is sensitive, an action occurs. By default, all widgets are sensitive when they are created.

You can set a widget to be nonsensitive using the *Nonsensitive* keyword in the WwSetValue function. When a widget is set to nonsensitive, its foreground color is grayed-out, and the widget cannot accept input from the user.

```
status = WwSetValue(widget, /Nonsensitive)
```

To change a widget from nonsensitive to sensitive, use the *Sensitive* keyword:

```
status = WwSetValue(widget, /Sensitive)
```

To determine if a widget is sensitive, use the *Sensitive* keyword in the WwGetValue function.

```
shown = WwGetValue(widget, /Sensitive)
```

If the widget is sensitive, the function returns 1; if nonsensitive, 0 is returned.
Displaying Widgets and Processing Events

After all of the widgets in a widget hierarchy have been created, they are displayed when the top-level or “root” window is displayed. The following command accomplishes this:

```wvave
status = WwSetValue(top, /Display)
```

For more information on the widget hierarchy, see The Widget Hierarchy on page 166.

Next, control must be transferred to the main event loop, which handles events (mouse clicks, for example) and executes callbacks. To do this, simply call the WwLoop function:

```wvave
WwLoop
```

The application remains in the main loop until the top-level window is closed with the following call:

```wvave
status = WwSetValue(top, /Close)
```

or until the top-level window is closed from the window manager menu. Under Motif, the window manager menu usually contains a Close button.

While the loop is running, any callback procedures that have been defined are executed whenever the appropriate events occur.

The use of WwSetValue to display and close widgets is demonstrated in examples throughout this chapter. See for example File Selection Example on page 197.

Programming Tips and Cautions

PV-WAVE Routines to Avoid

Avoid using the following routines in applications developed with WAVE Widgets or the Widget Toolbox. These routines wait for keyboard input and thus block the GUI. Where possible, alternative methods are suggested.

**Standard Library Routines**

- GET_KBRD — Try using a text field widget instead.
- HAK — Try using a non-blocking message widget instead.
- MOVIE — Try using WgMovieTool instead.
User Library Routines

- ANMENU — Try using WAVE Widgets menus instead.
- UNCMPRS_IMAGES
- XANIMATE — Try using WgAnimateTool instead.

PV-WAVE Routines to Use with Caution

Use the following routines with caution in applications developed with WAVE Widgets or the Widget Toolbox. All of these routines block the user from interacting with the GUI.

Standard Library Routines

- C_EDIT — Try using WgCeditTool instead.
- COLOR_EDIT — Try using WgCeditTool instead.
- CURSOR — Try using the Widget Toolbox event handler instead.
- DEFROI
- PALETTE — Try using WgCeditTool instead.
- PROFILES
- RDPIX
- WAIT — Try using the WtTimer function instead.
- WMENU — Try using a menu bar widget instead.
- ZOOM

Application Example

The following example program uses WAVE Widgets to create an image processing application that includes a drawing area for displaying the image and a menu containing image processing functions. Figure 5-30 on page 216 shows the main window of the application with an image displayed in the drawing area.

You can find this program file in the following location:

Under UNIX

$WAVE_DIR/demo/wavewidgets/simple_image.pro
Under OpenVMS

WAVE_DIR: [DEMO.WAVEWIDGETS] SIMPLE_IMAGE.PRO

; Example of WAVE Widgets. It displays an image, and allows you to
; change color table, and do some basic Image Processing.
;
PRO FileOK, wid, shell ; File Selection is done, let's load it
common widgets, top, slider
common images, orig, image, draw, size
common rotate, sliderval

    file = WwGetValue(wid) ; Get the file name
    file = FINDFILE(file) ; Find the file

    if N_ELEMENTS(file) lt 1 then begin ; File not found display warning
        message=WwMessage(wid,'File Not Found!'/WARNING,TITLE='File Error')
        endif else begin ; Got a file lets load it

            OPENR, /GET_LUN, unit, file(0)
            status=FSTAT(unit) ; Get the size and hope it is square image
            size=LONG(SQRT(status.SIZE)) ; Calculate height,width
            image=MAKE_ARRAY(size,size,/BYTE); make new image array
            READU,unit,image ; Read image, close the unit
            FREE_LUN, unit
            orig=image ; Store original for reload
            sliderval = 0 ; Reste slider value
            status=WwSetValue(slider,0) ; Reset rotate slider
            status=WwSetValue(draw,[size,size]) ; Set new draw area value
            DrawCB, draw, 1 ; Redraw the image

        endelse

        status = WwSetValue(shell,/CLOSE) ; Close the file selection window
END

PRO FileCancel, wid, shell ; File selection canceled
    status = WwSetValue(shell,/CLOSE) ; Close the file selection window
END

PRO FileCB, wid, which ; File handling
common widgets, top, slider
common images, orig, image, draw, size

    case which of
        1: begin ; Reload the image
            image=orig
            status=WwSetValue(slider,0) ; Reset rotate slider
            DrawCB,draw,1
2: begin ; Display file selection window
    if !version.platform eq 'vms' then $
        dir = getenv('WAVE_DIR')+'[data]' $
    else $
        dir = getenv('WAVE_DIR')+'/data/'$
    file = WwFileSelection(wid,'FileOK','FileCancel',$
        POSITION=[200,200],$
        TITLE='Load Image',DIR=dir,PATTERN='*.img')$
end

3: status=WwSetValue(top,/Close) ; Close the application, Bye, bye.
endcase

END

PRO ImageCB, wid, which; Modifying image
common images, orig, image, draw, size
    case which of
        1: image=ERODE(image,[[0,1,0],[1,1,1],[0,1,0]],/Gray) ; Erode
        2: image=DILATE(image,[[0,1,0],[1,1,1],[0,1,0]],/Gray) ; Dilate
        3: image=SHIFT(ALOG(ABS(FFT(image,-1))),size/2) ; FFT
        4: image=HIST_EQUAL(image) ; Histogram
        5: image=ROBERTS(image) ; Roberts
        6: image=SMOOTH(image,5) ; Smooth
        7: image=SOBEL(image) ; Sobel
    endcase
    DrawCB, draw, 1 ; Redraw the image
END

PRO ColorCB, wid, which ; Loading new colorable
    LOADCT,which-1,/SILENT
END

PRO DrawCB, wid, windex; Drawing the image
common images, orig, image, draw, size
    TVSCL, image
END

PRO SliderCB, wid, which ; Lets rotate the image
common images, orig, image, draw, size
    common rotate, sliderval
    value=WwGetValue(wid)
    if sliderval ne value then begin
        image=ROT(image,value)
DrawCB, draw, 1 ; Redraw the image  
sliderval = value  
endif  
END

PRO simple_image

common widgets, top, slider  
common images, orig, image, draw, size  
common rotate, sliderval  
image = Bytarr(512,512)  
size=512  
sliderval=0

; Loading first image  
if !version.platform eq 'vms' then$
   filename=getenv('WAVE_DIR')+'[data]head.img'$
else$
   filename='$WAVE_DIR/data/head.img'$

  test=FINDFILE(filename)
  if N_ELEMENTS(test) lt 1 then begin
     if test(0) eq '' then begin
        message,'Data subdirectory not available'
        EXIT
     endif
  endif
endif

Openr,1,filename
readu,1,image
close, 1
orig=image

top=WwInit('simple_image','Examples',layout,BACKGR='SkyBlue',$
   POSITION=[100,100], /VERTICAL, SPACING=5)

; Main menu bar for File, Image Processing, Color Tables  
menus={,$
   menubutton:'File',$
   menu:{,callback:'FileCB',title:'File',$
      button:'Reload Image',$
      button:'Load New Image',$
      button:'Exit'},$  
   menubutton:'Image Processing',$
   menu:{,callback:'ImageCB',title:'Image',$
      button:'Erode',$
      button:'Dilate',$
      button:'FFT',$
      button:'Histogram',}$
}
button:'Roberts',$
button:'Smooth',$
button:'Sobel'},$
menubutton:'Color Tables',$
menu:{,callback:'ColorCB',title:'Color',$
button:'Black/White Linear',$
button:'Blue/White',$
button:'Green/Red/Blue/White',$
button:'Red Temperature',$
button:'Blue/Green/Red/Yellow',$
button:'Standard Gamma-II',$
button:'Prism',$
button:'Red/Purple',$
button:'Green/White Linear',$
button:'Green/White Exponential',$
button:'Green/Pink',$
button:'Blue/Red',$
button:'16 Level',$
button:'16 Level II',$
button:'Steps',$
button:'PV WAVE'}$
}

bar=WwMenuBar(layout, menus); Let's create menu bar

draw=WwDrawing(layout,1,'DrawCB',[400,400],[512,512]) ;Creating Draw Area

; Creating Slider for Rotation
slider = WwControlsBox(layout,'Rotate',[0,360],'SliderCB',/VERTICAL,$
/TEXT, FOREGROUND='red',BACKGROUND='yellow',WIDTH=300)

status=WwSetValue(top,/DISPLAY); Displaying widget hierarchy

WwLoop ; Waiting for callbacks
END

215
Figure 5-30  The main window of the image processing application simple_image.pro (Motif style).
Using the Widget Toolbox

Introduction to the Widget Toolbox

The Widget Toolbox is an application programmer’s interface (API) used to create graphical user interface (GUI) applications for PV-WAVE. The Widget Toolbox provides a high-level method of creating and manipulating the GUI, while using the flexibility and power of PV-WAVE to process and display data. The Widget Toolbox consists of system routines that give you access to all the widget types supported by the OSF Motif toolkit.

Windows USERS Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

This chapter describes the Widget Toolbox and the basic steps in incorporating widgets into PV-WAVE applications. The topics include:

- An overview of the Widget Toolbox
- How to use the Widget Toolbox
- A brief description of include files that are used with the Widget Toolbox
- An example application using the Widget Toolbox
The Widget Toolbox routines are primarily for PV-WAVE developers who are familiar with Xt Intrinsics/Motif-based programming or Windows programming. If you are not familiar with Motif or Windows programming, then use the WAVE Widgets functions described in Chapter 5, Using WAVE Widgets.

**NOTE** Visual Numerics does not support the use of Widget Toolbox applications with 24-bit hardware.

---

### Basic Steps in Creating the GUI

The basic steps involved in creating an application GUI with the Widget Toolbox are:

- Initialize the Widget Toolbox with a call to the WtInit function.
- Create widgets with the WtCreate function, and set resources that control the appearance and other characteristics of the widgets.
- Manage, display, and destroy widgets with the WtSet function.
- Add callbacks, event handlers, and timers.
- Run the application.
- Close the Widget Toolbox.

### Combining WAVE Widgets and Widget Toolbox Functions

It is possible to combine WAVE Widgets and Widget Toolbox functions in the same application.

Widget IDs returned by WAVE Widget routines can be used in Widget Toolbox routines, and Widget Toolbox widget IDs can be passed to WAVE Widget routines. The basic steps for creating an application that combines the two kinds of widget functions do not change. You must initialize either WAVE Widgets or the Widget Toolbox (WwInit or WtInit function), create the widgets, display or “realize” the widgets, and execute the main loop with either the WwLoop or WtLoop function. For information on WAVE Widgets, see Chapter 5, Using WAVE Widgets.
Initializing the Widget Toolbox

The PV-WAVE system function WtInit initializes the Widget Toolbox. You must execute WwInit before any other Widget Toolbox functions. WtInit does the following:

- Establishes a connection to the X server.
- Initializes Xt Intrinsics
- Initializes the Motif Toolkit/Windows Advanced Controls Library
- Initializes the Widget Toolbox.

For example:

```c
top = WtInit(name, class)
```

The `name` parameter specifies the name of the application, and the `class` parameter indicates a more general category of application class. For example, in the example program at the end of this chapter, the call to WtInit is:

```c
war(0) = WtInit('example', 'Examples')
```

where `example` is the name of the application, and `Examples` specifies a general class to which `example` belongs. One purpose of the general class (`Examples`) is that resources in a single resource file can be shared among elements of that class.

See also the example Widget Toolbox application at the end of this chapter.

Creating Widgets

Rather than dealing directly with X library windows, applications using Xt Intrinsics-based toolkits (Motif) use widgets.

A widget is a complex data structure containing interface-related data and set of procedures that perform actions on that data.

Each widget is represented externally by a widget ID. Widgets form hierarchies known as widget trees. The root of every widget hierarchy is a special type of widget called a shell. The shell widget provides an interface between the child widget and the window manager.

The WtCreate function provides the general mechanism for creating all PV-WAVE widgets. For example:

```c
result = WtCreate('name', class, parent, args)
```
Windows USERS  WtCreate is not supported for Windows.

The name parameter is a string that identifies the widget. The class parameter is a widget class ID that specifies the type of widget to be created. Widget class IDs are long values defined in the Standard Library files wtxmclass.pro. The widget class IDs for Motif are also listed in Appendix A, Motif Widget Classes.

The parent parameter must be a widget ID of a widget that already exists. This can be a shell or any other type of widget that can have child widgets. The args parameter specifies values for resources used by the widget. See the next section for information on setting resources.

See also the example Widget Toolbox application at the end of this chapter.

Setting and Getting Resources

You can change the way a widget appears or behaves by specifying values for resources used by the widget.

Resources are specified using an unnamed structure with each tag name representing the resource name and each tag definition representing the resource value.

An unnamed structure has the following general definition:

\[ x = \{, tag\_name_1; tag\_def_1, tag\_name_n; tag\_def_n\} \]

For detailed information on unnamed structures, see Creating Unnamed Structures on page 92.

A resource name is a string indicating the type of resource. The resource name of a Widget Toolbox widget is derived directly from Motif resource names. Motif resource names are listed in the OSF/Motif Programmer’s Guide.

Remove the XmN prefix from the Motif resource names. For example:

<table>
<thead>
<tr>
<th>Motif Resource</th>
<th>Widget Toolbox Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>XmNx</td>
<td>x</td>
</tr>
<tr>
<td>XmNy</td>
<td>y</td>
</tr>
<tr>
<td>XmNlabel</td>
<td>label</td>
</tr>
<tr>
<td>XmNforeground</td>
<td>foreground</td>
</tr>
</tbody>
</table>
The data type of a resource’s value depends on the type of the resource.

**Example**

```plaintext
args={,x: 30, y: 50, label:'Done', foreground: 'red'}
; Create an unnamed structure containing resource names and values.
wid=wtcreate('button', xmPushButtonWidgetClass, parent, args)
; Create a pushbutton widget and use the args structure to
; specify its resources.
```

See also the example at the end of this chapter.

---

**Managing, Displaying, and Destroying Widgets**

Except for top-level shell widgets, all widgets must be “managed” by a parent widget. A widget’s parent manages the widget’s size and location, determines whether or not the widget is mapped (associated with an X window), and also controls the input focus of the widget.

By default all widgets are managed when created. To unmanage a widget after creation, use the WtSet function with the *Unmanage* keyword. For example:

```plaintext
status=WtSet(wid, /Unmanage)
```

To display a widget hierarchy, “realize” the shell widget of a hierarchy using the WtSet function with the *Realize* keyword. For example:

```plaintext
status=WtSet(shellid, /Realize)
```

To undisplay an individual widget, “unmanage” it using WtSet with the *Unmanage* keyword. For example:

```plaintext
status=WtSet(wid, /Unmanage)
```

To undisplay a whole widget hierarchy, unmanage the shell widget using WtSet with the *Unmanage* keyword. For example:

```plaintext
status=WtSet(shellid, /Unmanage)
```

To destroy a widget use WtSet with the *Destroy* keyword. For example:

```plaintext
status=WtSet(wid, /Destroy)
```

To destroy and close a whole widget hierarchy, use the WtClose function. For example:

```plaintext
status=WtClose(shellid)
```

See also the example Widget Toolbox application at the end of this chapter.
Adding Callbacks (Motif Only)

Most widgets provide “hooks” that call particular procedures when a predefined condition occurs. These hooks are known as callback lists and the procedures are called callbacks. To add a callback to a widget’s callback list, use the WtAddCallback function:

\[
status = \text{WtAddCallback}(\text{wid}, \text{reason}, \text{procedure}, [\text{client\_data}])
\]

**Windows USERS**  
WtAddCallback is not supported for Windows.

The \text{wid} parameter is the ID of the widget to add the callback to. The \text{reason} parameter is a string that specifies the callback list to which the callback routine (function or procedure) is to be added. The reason name is derived from the Motif reason name. Remove the XmN prefix from the Motif reason names. For example:

<table>
<thead>
<tr>
<th>Motif Reason</th>
<th>Widget Toolbox Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>XmNactivateCallback</td>
<td>activateCallback</td>
</tr>
</tbody>
</table>

The application can optionally use the \text{client\_data} parameter to specify some application-defined data to be passed to the callback procedure when the callback is invoked. If \text{client\_data} is a local variable (defined only in the current procedure), a copy of that variable is created and passed (passed by value). If the \text{client\_data} is a global variable (defined in a Common Block), it is passed by reference.

The form of every Widget Toolbox callback procedure is:

\[
\text{PRO \ CallbackProc, widget, client\_data, } nparams, [p1, p2, ... pn]
\]

where:

\text{widget} — The widget ID.

\text{client\_data} — The \text{client\_data} passed to WtAddCallback.

\text{nparams} — The number of callback-specific parameters after \text{nparams}. Two additional parameters are required: \text{event} and \text{reason}. For information on these additional parameters, see Appendix B, Motif Callback Parameters.

\text{p} — The optional callback-specific parameters. For additional information on these parameters, see Appendix B, Motif Callback Parameters.
**Example**

This example adds a callback called `quitit` to the callback list for the widget `warr(2)`. The callback reason is `activateCallback`.

```plaintext
status=WtAddCallback(warr(2), 'activateCallback', 'quitit')
```

See also the example Widget Toolbox application at the end of this chapter.

---

**Adding Event Handlers**

An event handler is a procedure that is executed when a specific type of event occurs within a widget. Some, all, or no X events can be handled using one or more event handlers. To register an event handler for events that occur in a widget use the system function `WtAddHandler`:

```plaintext
status = WtAddHandler(wid, eventmask, handler, [client_data])
```

This function registers a callback procedure specified by the `handler` parameter (a string) as an event handler for the events specified by the `eventmask` parameter. The `eventmask` parameter must be one of the standard event masks defined in the file `wtxlib.pro` in the Standard Library. The `wid` parameter is the ID of the widget to add the handler to.

You can register event handlers for multiple event masks using the OR operator. For example, `ButtonUp` and `ButtonDown` are combined with `ButtonUp OR ButtonDown`, causing the event to be triggered when the mouse button is pressed and when it is released.

The application can optionally use the `client_data` parameter to specify some application-defined data to be passed to the event handler procedure when the callback is invoked. If `client_data` is a local variable (defined only in the current procedure), a copy of that variable is created and passed (by value). If `client_data` is a global variable (defined in a Common Block), it is passed by reference.

The form of Widget Toolbox event handlers is:

```plaintext
PRO EventHandlerProc, widget, client_data, $
    nparams, eventmask, event

where:

widget — The widget ID.

client_data — The `client_data` passed to WtAddHandler.
```
**nparams** — The number of event handler-specific parameters after `nparams`. (This number is always 2, for the `eventmask` and `event` parameters.)


**event** — Structure containing all the fields as defined for the Xlib XEvent structure. If you are developing under Motif, see Appendix E, “Event Reference”, of the *Xlib Reference Manual, Volume 2*, (O’Reilly & Associates, Inc., 1989) for a description of event structures. If you are developing under Windows, see *WtAddHandler Function* on page 365 for a description of the event structure.

**Example**

```plaintext
.
.
pane=WtCreate('menu', PopupMenuWidget, parent)
status = WtAddHandler(pane, ButtonPressMask, 'PostMenu', parent)
.
.
PRO PostMenu, wid, parent, nparams, mask, event
@wtxlib
status=WtPointer("GetLocation", wid, state)
if (Button3Mask AND state(6)) ne 0 then $
  status=WtSet(pane, POPUP=event)
END
```

**X Event Handler Procedure Example**

```plaintext
PRO handler, widget, data, nparams, mask, event
...
END
```

**Adding Timers**

A timer (the Xt Intrinsics term is TimeOut) is a procedure that is invoked when a specified time interval has elapsed. This function can be used to add or remove a
timer. To register a timer routine for a specified interval, use the WtTimer function in the application with the ADD parameter.

\[
\text{timerid} = \text{WtTimer("ADD", interval, timer, [client_data])}
\]

The interval parameter specifies the time interval, in milliseconds, until the PV-WAVE procedure timer callback timer is invoked. The WtTimer routine, unlike the Xt TimeOut function, restarts itself when the timer callback procedure is called. To stop the timer, use the following command in the timer callback:

\[
\text{status} = \text{WtTimer("REMOVE", timerid)}
\]

The application can optionally use the client_data procedure to specify some application-defined data to be passed to the timer callback procedure when the callback is invoked. If client_data is a local variable (defined only in the current procedure), a copy of that variable is created and passed (passed by value). If client_data is a global variable (defined in a Common Block), it is passed by reference.

The form of Widget Toolbox timer procedures is:

\[
\text{PRO TimerProc, widget, client_data, nparms, timerid, time}
\]

where:

- \text{widget} — The top application shell widget ID.
- \text{client_data} — The client_data passed to WtTimer.
- \text{nparms} — The number of timer-specific parameters after nparms. (This number is always 2, for timerid and time.)
- \text{timerid} — The unique timer ID.
- \text{time} — The time interval in milliseconds.

\underline{NOTE} This timer routine is not related to the TIMER procedure in the Users’ Library.

\section{Example}

common timer, tid
.
.
id=WtTimer("ADD", 100, 'TimerCallback', my_data)

\text{PRO TimerCallback, wid, client_data, nparms, timer_id, interval}
\text{COMMON timer, tid}
Adding Work Procedures

Most applications spend most of their time waiting for events to occur. You can register a work procedure that will be called when the toolkit is idle (waiting for an event). The work procedure is the only means offered by the Xt Toolkit for performing background processing. A work procedure is useful if you need to execute a time-consuming operation from a callback procedure.

When a work procedure is added, it is executed in its entirety unless a REMOVE call to WtWorkProc is issued before the procedure has been called.

If the work procedure does a large amount of processing it could block the widget interface until it has finished running.

A typical implementation using WtWorkProc to perform a large amount of background processing is to break down the processing into a number of discrete steps and execute the steps one at a time in a work procedure which issues another ADD call to WtWorkProc to start the next step. Repeat this process until all the steps have been completed.

To register a work procedure, use the system function WtWorkProc:

\[
status = \text{WtWorkProc}\left(\text{function, parameters}\right)
\]

The function parameter is an Add or Remove operation. Add registers a named work procedure. The parameters used depend on whether an Add or Remove operation is specified.

Adding Input Handler Procedures (Motif Only)

While most GUI applications are driven only by events, some applications need to incorporate other sources of input into the X Toolkit event handling mechanism. WtInput supports input or output gathering from files. The application registers an input source handler procedure and a file with the X Toolkit. When input is pending on the file, the registered handler is invoked. Note that a “file” in this context should be loosely interpreted to mean any sink (destination of output) or pipe (source of data).
To register an input handler procedure, use the system function WtInput:

\[
\text{status} = \text{WtInput}(\text{function \{, \parameters})}
\]

The \textit{function} parameter is an Add or Remove operation. Add registers an input handler procedure. The \textit{parameters} used depend on whether an Add or Remove operation is specified.

The form of a Widget Toolbox input handler procedure is:

\begin{verbatim}
PRO InputHandlerProc, widget, client_data, $
    nparms, inputid, lun, source
\end{verbatim}

where:

\begin{itemize}
    \item \textit{widget} — The top application shell widget ID.
    \item \textit{client_data} — The client data passed to WtInput.
    \item \textit{nparms} — The number of input handler-specific parameters after \textit{nparms}. This number is always three, for the \textit{inputid}, \textit{lun}, and \textit{source} parameters.
    \item \textit{inputid} — A unique input handler ID.
    \item \textit{lun} — The logical unit number of the source (could be a file) generating the event.
    \item \textit{source} — The file descriptor of the source (could be a file) generating the event.
\end{itemize}

For more information on WtInput, see \textit{WtInput Function (Motif Only)} on page 381.

---

\section*{Changing the Cursor}

The WtCursor function lets you change or set the cursor. For instance, when a long file is read into memory, you can display a wait cursor. You can select from a large number of cursors listed in Appendix C, Widget Toolbox Cursors.

A call to WtCursor has the following form:

\[
\text{status} = \text{WtCursor}(\text{function, widget \{, index})}
\]

The \textit{function} parameter specifies the type of cursor: default, system, wait, or set. If set is specified, then you must specify a cursor index from the list in Appendix C, Widget Toolbox Cursors.
Creating Tables

The WtTable function lets you modify tables of data created with the XbaeMatrix widget. The XbaeMatrix widget is an editable 2D array of string data (cells) similar to a spreadsheet.

The Motif version of the XbaeMatrix widget was originally developed by Andrew Wason of Bellcore.

Complete documentation for the XbaeMatrix widget is available in the PostScript file matrix_motif.ps, which you can print on any PostScript printer. This file is in:

(UNIX) $VNI_DIR/wave/docs/widgets
(OpenVMS) VNI_DIR: [WAVE.DOCS.WIDGETS]
(Windows) %VNI_DIR\wave\docs\widgets

Where VNI_DIR is the main Visual Numerics directory.

Refer to this document for detailed information on the MbaeMatrix widget’s resources and callbacks.

Running an Application

When all the widgets to be displayed are created and managed, and callbacks, handlers, and timers are defined, you must then realize (display) the root widget in the widget hierarchy and initiate the event loop:

```
top = WtInit('appl','Appl')

; Create the widgets.

status = WtSet(top, /Realize)

; Display the widget hierarchy.
WtLoop

; Initiate the event loop.
```

This causes the routine to loop indefinitely, processing the events and dispatching callbacks, handlers, and timers. The WtLoop procedure can be stopped by destroying and closing the shell widget by calling WtClose:

```
status=WtClose(top)
```
Related Include Files

The following files are located in the PV-WAVE Standard Library. They are used in Widget Toolbox applications as include files. To include a file in a program, use the @ command. For example:

@wtxmclasses

- wtxmclasses.pro — Contains definitions of Motif widget classes. Include this routine in every procedure that creates or handles Motif widgets.
- wtxmconsts.pro — Contains definitions of Motif-related constants (for example, Xm...). Include this routine in each procedure using Motif-related constants.
- wtxlib.pro — Contains X Event mask and type definitions, and other Xlib constants for X Event handling. Include this file whenever you need to use these constants.
- wtcursor.pro — Contains the indexes for standard and custom cursors.

Example Widget Toolbox Application

The following program demonstrates how to use Widget Toolbox functions to create a simple GUI. This program displays a form containing a single button. When you click on the button, the button’s label changes. The following figure shows the example program’s output:
Figure 6-1 Motif GUI created by the example program.

```
PRO QUITIT, wid, data, npars, reason, event, count
    ; The QUITIT procedure is a callback routine that is executed
    ; when the "Hello World" button is selected.
    COMMON widgets, warr, pushed
    ; If the button is selected, replace the button label with "Good Bye!!!".
    ; Next time the button is selected, close the top-level window to quit
    ; the application.
    IF pushed EQ 0 THEN BEGIN
        args={,labelString:'Good Bye!!!!'}
        ; args is defined as an unnamed structure.
        status=WtSet(wid,args)
        pushed=1
    ENDIF ELSE BEGIN
        PRINT, 'Quitting...
        status=WtClose(warr(0))
    ENDElse
END

PRO EXAMPLE
    ; This procedure creates a pushbutton with the QUITIT routine as a the callback.
    @wxmclasses
    @wxmconsts
    COMMON widgets, warr, pushed
    warr = LONARR(4)
    pushed=0
    ; Initialize the application.
    warr(0)=WtInit('example','Examples')
    ; Create a "container" widget.
    warr(1) = WtCreate('form',xmFormWidgetClass,$
        warr(0))
    args={,x:10,y:10}
    ; Create a label.
    warr(2)=WtCreate('Push Me, Please!',$ 
        xmLabelWidgetClass, warr(1),args)
    args={,y:40,width:140,height:25,recomputeSize:FALSE}
    ; Create the "Hello World" push button.
    warr(3)=WtCreate('Hello,World!',$
        xmPushButtonWidgetClass, warr(1),args)
```
; Add the button callback.
status=WtAddCallback(warr(3), 'activateCallback', 'quitit')

; Display the top-level shell.

status=WtSet(warr(0), /Realize)

; Process events. Call callbacks.
WtLoop
END

Programming Tips and Cautions

For information on routines to avoid or use with caution when you are developing widgets applications, see Programming Tips and Cautions on page 210.

See also Appendix D, Developing Portable Applications.
WAVE Widgets Reference

WwAlert Function

Creates a modal (blocking) or modeless (non-blocking) popup alert box containing a message and optional control buttons.

Usage

\[
\text{wid} = \text{WwAlert}(\text{parent}, \text{label} [, \text{answers}])
\]

Input Parameters

- **parent** — The widget ID of the parent widget.
- **label** — A string containing the message text.
- **answers** — (optional) A string or array of strings (0 – 3) containing the text to appear on the buttons. If this parameter is not specified, the following default text is used: OK, Cancel, and Help.

Returned Value

- **wid** — The index number of the button that was clicked. If the *Nowait* keyword is specified, the alert box ID is returned.
Keywords

After — Specifies the number of seconds that the alert box appears. After the specified number of seconds, the alert box is automatically destroyed, and WwAlert returns the value –1.

Block — If nonzero, creates a modal (blocking) alert box.

Error — If nonzero, creates an error alert box.

Help — A string specifying the name of a topic in the online Help system (i.e., Hyperhelp on UNIX and OpenVMS platforms or Windows Help on Windows platforms). For a list of valid Help topics, bring up the Search dialog box from the main online Help window.

Helpfile — Specifies a string containing the name of an online Help file.

Info — If nonzero, creates an information alert box.

Name — Specifies a string containing the name for the Warning, Question, Working, or Information dialog widget. (Default: Alert.)

NoConfirm — If nonzero, no buttons are displayed in the alert box.

NonBlock — If nonzero, creates a non-blocking alert box. (This keyword has no effect under Microsoft Windows.)

Nowait — If nonzero, WwAlert returns to the caller immediately after the alert box appears, and WwAlert returns the alert box ID.

Question — If nonzero, creates a question alert box.

Raise — If nonzero, the alert box always appears in front of other windows.

Title — Specifies a string containing the alert box title.

Warning — If nonzero, creates a warning alert box.

Working — If nonzero, creates a working alert box.

Color/Font Keywords

Background — Specifies the background color name.

Font — Specifies the name of the font used for the text that appears in the alert box.

Foreground — Specifies the foreground color name.

Get/Set Value

None.
**Callback Parameters**

None.

**Discussion**

The alert box can be used to:

- Halt execution of the program until the alert condition is satisfied.
- Allow the calling routine to display an alert with a specified number of buttons (0 – 3) and button labels.

The alert box supports the following features:

- Multiline labels. (Use the ASCII code \012 at the end of each line to indicate a \textlt{Return}, or you can use an array of strings, where each array element is taken to be a separate line.)
- Context sensitive help.

**NOTE** If the \textit{Nowait} keyword is specified, control is returned to the calling routine immediately after the alert becomes visible. After that point, it is up to the calling routine to destroy the alert box with the \texttt{WwAlertPopdown} function.

**Example**

This example creates a button box containing seven buttons that test the features of the \texttt{WwAlert} function. The callback \texttt{ButtonCB} is executed when one of the buttons in the button box is selected.

Enter the callback procedure and the main procedure into a file, and compile it with the .RUN command. To run the example, enter \texttt{testalert} at the \texttt{WAVE}> prompt. To dismiss the button box, select the appropriate function (such as Close) from the window manager menu.

```pro
PRO ButtonCB, wid, idx
  message = ['This is an example', $ 'of the Alert Box', 'Wave Widget']
  CASE idx OF
    1:button=WwAlert(wid, message)
    2:button=WwAlert(wid, message, After = 5)
    3:button=WwAlert(wid, message, /Noconfirm, /Raise)
    4:button=WwAlert(wid, message, "Done")
    5:button=WwAlert(wid, message, ["Done","Dismiss"])
```

\textbf{WwAlert Function 235}
6: button=WwAlert(wid, message, ["Done","Dismiss","Bye"])
7: button=WwAlert(wid, message, /Noconfirm, /Nowait)
ENDCASE
PRINT, 'Button selected: ', button
CASE button OF
  1: PRINT, 'Selected OK'
  2: PRINT, 'Selected Cancel'
  3: PRINT, 'Selected Help'
ELSE: PRINT, 'None selected'
ENDCASE
IF idx EQ 7 THEN BEGIN
  WAIT, 5
  PRINT, 'Done waiting, popping it down...
  WwAlertPopdown, button
ENDIF
END
PRO testalert
  top = WwInit('testalert', 'Test', layout)
  buttons = ['Alert', 'Alert After', $'
    0 button', '1 button', '2 buttons', $'
    3 buttons', 'Nowait']
  button = WwButtonBox(layout, buttons, "ButtonCB")
  status = WwSetValue(top, /Display)
  WwLoop
END

See Also

WwAlertPopdown, WwMessage
**WwAlertPopdown Procedure**

Destroys an alert box.

**Usage**

WwAlertPopdown, *wid*

**Input Parameters**

*wid* — The widget ID of the alert box to be destroyed.

**Keywords**

None.

**Discussion**

This function is used to destroy an alert box that was created when the WwAlert *Nowait* keyword was specified.

**Example**

For an example of WwAlertPopdown, see the WwAlert function example.

**See Also**

WwAlert
WwButtonBox Function

Creates a horizontally or vertically oriented box containing push buttons.

Usage

\[bbox = \text{WwButtonBox}(\text{parent}, [\text{labels},] \text{callback})\]

Input Parameters

\textit{parent} — The parent widget’s ID.

\textit{labels} — (optional) A string or an array of strings containing the text that is to appear on the buttons. If the \textit{labels} parameter is defined as a null string or an array of null strings, the function looks for button labels in a resource specification (see Discussion).

\textit{callback} — A string containing the name of the callback routine.

Returned Value

\textit{bbox} — The ID of the button box widget.

Input Keywords

\textit{Border} — Specifies the width in pixels of the button box and button borders.

\textit{Center} — An array specifying the position of the left and right edge of buttons as a percentage of button box width. By default, buttons are spaced evenly in the box.

\textit{Form} — When present and nonzero, buttons are placed in a form layout and all specified attachment keywords are honored (i.e., /Left, /Right, /Top, /Bottom). By default, buttons are placed in a row/column layout.

\textit{Horizontal} — If present and nonzero, creates a horizontally aligned row of buttons.

\textit{Layout_name} — Specifies the name of the container widget used to organize the buttons. The \textit{Layout_name} specified is the compound widget layout name used in the resource specification. (Default: buttons.)

\textit{Measure} — Specifies the number of columns of buttons (for a vertical box) or rows (for a horizontal box).

\textit{Name} — Specifies an array of strings containing the names of the button widgets in a resource specification. The \textit{Name} keyword can be used in place of the \textit{labels}
parameter, although *labels* (if other than an array of null strings) will take precedence if both are given. (Default: button_0, button_1, ..., button_n.)

**Position** — If the button box widget is to be placed in a bulletin board layout, use this keyword to specify the x, y coordinates of the button box widget within the bulletin board.

**Vertical** — When present and nonzero, creates a vertically aligned column of buttons.

**Spacing** — Specifies the space in pixels between buttons.

### Output Keywords

**Buttons** — Returns an array of push button widget IDs.

### Color/Font Keywords

**Background** — Specifies the background color name. Background color is the color of the button.

**Basecolor** — Specifies the base color of the container widget.

**Font** — Specifies the name of the font used for button text.

**Foreground** — Specifies the foreground color name. Foreground color is the color of the button text.

### Attachment Keywords

**Bottom** — If a widget ID is specified (for example, Bottom=wid), then the bottom of the button box widget is attached to the top of the specified widget. If no widget ID is specified (for example, /Bottom), then the bottom of the button box widget is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, Left=wid), then the left side of the button box widget is attached to the right side of the specified widget. If no widget ID is specified (for example, /Left), then the left side of the button box widget is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, Right=wid), then the right side of the button box widget is attached to the left side of the specified widget. If no widget ID is specified (for example, /Right), then the right side of the button box widget is attached to the right side of the parent widget.
Top — If a widget ID is specified (for example, Top=wid), then the top of the button box widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, /Top), then the top of the button box widget is attached to the top of the parent widget.

Get/Set Value

getvalue — Gets the label of the selected button.

setvalue — Sets the label of the selected button.

Callback Parameters

Any button box callback procedure must have the following two parameters:

wid — The button widget ID.

index — Index of the button pushed (1 – n).

Discussion

The “button box” is a special widget in which individual buttons are arranged. If only one button is requested, that button’s widget ID is returned. By default, the buttons are arranged in a row/column format. See the WwLayout function for information on row/column format.

Part of the button box widget resource names can be specified using the Name keyword, otherwise the defaults are the *.button_#.labelString resources.

TIP The labels parameter provides a method for “hard-coding” the button names in the application. For greater flexibility, create your resource files using a text editor, and load the resources containing the button names using WtResource. The Name keyword can then be used in the WwButtonBox calling sequence to specify the names for the button widgets.

Example 1

This example creates a button box containing three buttons, Quit, Dialog, and Message. The callback ButtonCB is executed when one of the buttons in the button box is selected.

Enter the callback procedure into a file, and compile the procedure with the .RUN command. Then, enter the widget commands at the WAVE> prompt. To dismiss the
button box, select the appropriate function (such as **Close**) from the window manager menu.

**Callback Procedure**

```pro
PRO ButtonCB, wid, data
CASE data OF
  1: PRINT,'Quit Selected'
  2: PRINT,'Dialog Selected'
  3: PRINT,'Message Selected'
ENDCASE
END
```

**Widget Commands**

```pro
labels = ['Quit','Dialog','Message']
top=WwInit('ww_ex20', 'Examples', layout)
bbox=WwButtonBox(layout, labels,'ButtonCB', /Horizontal, $ Center=[10,30,40,60,70,100])
status=WwSetValue(top, /Display)
WwLoop
```

**Example 2**

A typical resource specification for the button names used in WwButtonBox is:

```pro
myapp.layout.buttonform.quit_button.labelString: Quit
```

**See Also**

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, *Using WAVE Widgets*.

For additional information on the color and font keywords, see *Setting Colors and Fonts* on page 201.

For additional information on attachment keywords, see *Form Layout: Attachments* on page 170.

For information on Get and Set values, see *Setting and Getting Widget Values* on page 206.
**WwCallback Function (Motif Only)**

Adds or removes a WAVE Widgets callback.

**Usage**

```
status = WwCallback(wid, callback, reason, client_data)
```

**Input Parameters**

- `wid` — The ID of the widget for which to add or remove the callback.
- `callback` — A string containing the name of the PV-WAVE callback routine.
- `reason` — A string containing the callback reason. (See the *Discussion* section for more information.)
- `client_data` — A variable. The value of this variable is passed to the callback routine.

**Returned Value**

- `status` — A value indicating success or failure of the addition or removal of the WAVE Widgets callback.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Indicates success.</td>
</tr>
<tr>
<td>0</td>
<td>Indicates failure.</td>
</tr>
</tbody>
</table>

**Keywords**

- `Add` — If nonzero, the specified callback is added.
- `Params` — If nonzero, the callback is called with all parameters. (Default: called with only the `wid` and `data` parameters; see the *Callback Parameters* section)
- `Remove` — If nonzero, the specified callback is removed.

**Callback Parameters**

Any added callback procedure must have the following two parameters:

- `wid` — The widget ID.
- `data` — User-defined data.
Discussion

Callback reasons are listed throughout the *OSF/Motif Programmer’s Reference*.

Windows USERS  Visual Numerics has ported a subset of the Motif widget functionality to Microsoft Windows. To use WwCallback, you must refer to the *OSF/Motif Programmer’s Reference* to obtain callback reasons. Because Motif widgets are only partly implemented for Windows, some callback reasons are not available for use in Windows applications.

To use a Motif callback reason in PV-WAVE, remove the XmN or XtN prefix. For example:

<table>
<thead>
<tr>
<th>Motif Reason</th>
<th>WAVE Widget Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>XmNactivateCallback</td>
<td>activateCallback</td>
</tr>
</tbody>
</table>

Use of the *client_data* parameter is optional in the application. It can specify some application-defined data to be passed to the callback procedure when the callback is invoked. If *client_data* is a local variable (defined only in the current procedure), a copy of that variable is created and passed by value. If *client_data* is a global variable (defined in a Common Block), it is passed by reference.

See Also

WtAddCallback

WwCommand Function

Creates a command window.

Usage

\[ command = \text{WwCommand}(parent, \text{enteredCallback, doneCallback}) \]

Input Parameters

*parent* — The parent widget’s ID.

*enteredCallback* — A string containing the name of the callback routine that is executed when a command is entered and confirmed (the user presses <Return>).
**doneCallback** — A string containing the name of the callback routine that is executed when the command window is destroyed.

**Returned Value**

**command** — The widget ID of the command window.

**Keywords**

**Maximum** — Specifies the maximum number of items that can be placed in the command history list.

**Name** — A string specifying the command widget name. This string is a part of a resource specification for the command window prompt. The **Name** keyword can be used in place of the **Prompt** keyword, although **Prompt** will take precedence if both are used. (Default: `command`.)

**Position** — Specifies the position of the upper-left corner of the command window on the screen.

**Prompt** — A string containing the prompt for the WwCommand widget.

**Shell_name** — A string specifying the name of the top-level widget, or TopLevelShell as part of the resource specification. (Default: `commandshell`.)

**Title** — Specifies a string containing the command window title.

**Visible** — Specifies the maximum number of command history items that are visible.

**Width** — Specifies the width of the command window.

**Color/Font Keywords**

**Background** — Specifies the background color name.

**Font** — Specifies the name of the font used for text.

**Foreground** — Specifies the foreground color name.

**Get/Set Value**

**getvalue** — Gets a string array containing the list of commands entered, from the oldest to newest command.

**setvalue** — Sets a new command in the text input field.
Callback Parameters

Any command widget callback procedure must have the following two parameters:

- **container** — Container widget ID.
- **wid** — Popup window widget ID.

Discussion

A command window is a *popup* window. This means that it cannot be the child of the top-level shell or the layout widget. Usually, a command widget is activated by a pushbutton or menu button, as in the example below.

A command window provides:

- A text input field where the user can enter text, such as a command or a label.
- A scrolling command-history list. Whenever the user enters a command in the text input field and presses <Return>, a callback is executed and the command text is placed on the history list. The user can re-enter a previously entered command by clicking on it in the command history list.

  If the user double-clicks on a command in the command history list, the callback is automatically executed and appended to the end of the list.

- A label for the text input field.

If *Prompt* is not used, the function looks for a command window prompt in a resource specification. Part of the resource specification can be specified using the *Name* keyword, otherwise the default is the *command.promptString* resource (where promptString is the attribute).

**TIP** The *Prompt* keyword provides a method for “hard-coding” the command window prompt in the application. For greater flexibility, create your resource file using a text editor, and load the resource containing the prompt string using WtResource. The *Name* keyword can then be used in the WwCommand calling sequence to specify the command widget name in the resource specification.

Example 1

This example creates a button labeled *Command*. When this button is selected, the *CbuttonCB* callback is activated and a command window is created. When a user enters text in the text input field and presses <Return>, the callback *CommandOK*...
is executed. When the user exits the command box, the CommandDone callback is executed.

Enter the callback procedures into a file, and compile the procedures with the .RUN command. Then, enter the widget commands at the WAVE> prompt. To dismiss the widgets, select the appropriate function (such as Close) from the window manager menu of the Command button (the parent widget).

**Callback Procedures**

```apl
PRO CbuttonCB, wid, data
  command = WwCommand(wid, 'CommandOK', $
                  'CommandDone', Position=[300,300], $
                  Title='Command Entry Window')
END
PRO CommandOK, wid, shell
  value = WwGetValue(wid)
  PRINT, value
END
PRO CommandDone, wid, shell
  status = WwSetValue(shell, /Close)
END
```

**Widget Commands**

```apl
top=WwInit('example2', 'Examples', layout)
button=WwButtonBox(layout, 'Command', 'CbuttonCB')
status=WwSetValue(top, /Display)
WwLoop
```

**Example 2**

A typical resource specification for the command window prompt used in WwCommand is:

```apl
myapp.commandshell.command.promptString: Enter a command:
```

**See Also**

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, *Using WAVE Widgets*. 

WwControlsBox Function

Creates a box containing sliders.

Usage

controls = WwControlsBox(parent, [labels,] range, changedCallback)

Input Parameters

parent — The widget ID of the parent widget.

labels — (optional) A string or array of strings containing the text used to label the sliders. If the labels parameter is not defined or is defined as a null string or an array of null strings, the function looks for slider labels in a resource specification (see Discussion).

range — An array of values specifying the minimum and maximum slider values.

changedCallback — A string containing the name of the callback routine that is executed when the value of a slider changes.

Returned Value

controls — The widget ID of the controls box widget.

Input Keywords

Border — Specifies the width in pixels of the controls box and slider borders.

Center — An array specifying the position of the left and right edge of sliders as a percentage of controls box width. By default, buttons are spaced evenly in the box.

Drag — If present and nonzero, the callback procedure is called while the slider is being dragged.

Float — Lets you display a floating-point slider, with ranges that include implied decimal numbers. This keyword specifies the number of digits that appear after the
decimal point. For example, to display a floating-point slider with the range 0.5 to 12.5, use:

controls=WwControlsBox(parent, ‘myslider’,[5, 125], $ ’mycallback’, Float=1)

The decimal point is implied in that it is used for display purposes only. If the user chooses a slider value of 5.7, the value returned to the callback is 57. The value of Float can be obtained through the Userdata keyword of the WwGetValue function. Thus, you can obtain the implied decimal position that was used in a WwControlsBox function. For example:

PRO mycallback, wid, which
   ndec = WwGetValue(wid, /Userdata)
   value = WwGetValue(wid)
   PRINT, ‘User Selected Value: ’, Float(value)/10.0^ndec

**NOTE** If you pass floating point values to the range parameter, be sure that you set the Float keyword as well. Otherwise, the value returned by the function will be unexpected.

*Form* — When present and nonzero, sliders are placed in a form layout and all specified attachment keywords are honored (i.e., /Left, /Right, /Top, /Bottom). By default, sliders are placed in a row/column layout.

*Height* — Specifies the height of the sliders. (Unix only.)

*Horizontal* — When present and nonzero, creates a horizontally aligned row of sliders.

*Hslider* — When present and nonzero, creates horizontally oriented sliders.

*Layout_name* — Specifies the name of the layout widget. This name is part of the resource specification. The default name for Layout_name depends on other keywords specified in the WwControlsBox usage.

   If *Form* or *Horizontal* or *Vertical* are used to organize the sliders, the Layout_name default is controls.

   If the *Form* widget is used to organize the slider and *Text* is also specified, the default for Layout_name is ctrlform.

*Measure* — Specifies the number of columns of sliders (for a vertical box) or rows (for a horizontal box).

*Name* — Specifies an array of strings containing the names of the Scale widget and, optionally, the TextField widget if *Text* is also specified. The Name keyword can be used in place of the labels parameter, although labels (if other than an array
of null strings) will take precedence if both are given.
(Default: slider_0, text_0, slider_1, text_1, ..., slider_n, text_n.)

**Position** — If the controls box widget is to be placed in a bulletin board layout, use this keyword to specify the \(x, y\) coordinates of the controls box widget within the bulletin board.

**Spacing** — Specifies the space between sliders.

**Text** — When present and nonzero, an input text field is created for each slider.

**Vertical** — When present and nonzero, creates a vertically aligned column of sliders.

**Vslider** — When present and nonzero, creates vertically oriented sliders.

**Width** — Specifies the width of the sliders. (Unix only.)

### Output Keywords

**Sliders** — Returns an array of slider widget IDs.

### Color/Font Keywords

**Background** — Specifies the background color name.

**Basecolor** — Specifies the base color.

**Font** — Specifies the name of the font used for slider text.

**Foreground** — Specifies the foreground color name.

### Attachment Keywords

**Bottom** — If a widget ID is specified (for example, `Bottom=wid`), then the bottom of the controls box widget is attached to the top of the specified widget. If no widget ID is specified (for example, `/Bottom`), then the bottom of the controls box widget is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, `Left=wid`), then the left side of the controls box widget is attached to the right side of the specified widget. If no widget ID is specified (for example, `/Left`), then the left side of the controls box widget is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, `Right=wid`), then the right side of the controls box widget is attached to the left side of the specified widget. If no
widget ID is specified (for example, /Right), then the right side of the controls box widget is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, Top=wid), then the top of the controls box widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, /Top), then the top of the controls box widget is attached to the top of the parent widget.

**Get/Set Value**

*getvalue* — Gets the value of the selected slider.

*setvalue* — Sets the value of the selected slider.

**Callback Parameters**

Any controls box callback procedure must have the following two parameters:

*wid* — Slider widget ID.

*index* — Index value of the slider that has changed (1 – n).

**Discussion**

A “controls box” is a special widget in which individual sliders are arranged. If only one slider is requested, that slider’s widget ID is returned. By default, the sliders are placed in a row/column format.

A slider allows the user to change a value interactively by moving the slider handle back and forth within a predefined range. You have the option of including a text input field with each slider. The text input field lets the user enter an exact value for the slider.

The widget name portion of the slider label resource specification can be specified using the *Name* keyword.

---

**TIP** The *labels* parameter provides a method for “hard-coding” the slider labels in the application. For greater flexibility, create your slider label resource file using a text editor, and load the resources containing the slider labels and text strings using WtResource. The *Name* keyword can then be used in the WwControlsBox calling sequence to specify the slider (and text) widget names.
**Example 1**

This example creates a box with three sliders labeled **Pressure**, **RPM**, and **Temperature**. Whenever the user moves one of the sliders, the callback procedure is executed.

Enter the callback procedure into a file, and compile the procedure with the `.RUN` command. Then, enter the widget commands at the `WAVE>` prompt. To dismiss the controls box, select the appropriate function (such as `Close`) from the window manager menu.

**Callback Procedure**

```plaintext
PRO SliderCB, wid, which
CASE which OF
  1: PRINT,'First Slider Moved'
  2: PRINT,'Second Slider Moved'
  3: PRINT,'Third Slider Moved'
ENDCASE
value = WwGetValue(wid)
PRINT, value
END
```

**Widget Commands**

```plaintext
top=WwInit('ww_ex22', 'Examples', layout)
labels=['Pressure','RPM','Temperature']
ranges=[0,100,2000,4000,50,150]
controls = WwControlsBox(layout, labels, $
  ranges, 'SliderCB',/Vertical,/Text, $
  Foreground='gray',Background='white', $
  Basecolor='blue')
status=WwSetValue(top, /Display)
WwLoop
```

**Example 2**

A typical resource specification used in `WwControlsBox` is:

```plaintext
myapp.layout.sliderform.xslider.titleString: X Rotation
```
See Also
For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, Using WAVE Widgets.
For additional information on the color and font keywords, see Setting Colors and Fonts on page 201.
For additional information on attachment keywords, see Form Layout: Attachments on page 170.
For information on Get and Set values, see Setting and Getting Widget Values on page 206.

WwDialog Function
Creates a blocking or nonblocking dialog box.

Usage

\[
wid = \text{WwDialog}(\text{parent}, [\text{label},] \ \text{OKCallback}, \ \text{CancelCallback}, \ \text{HelpCallback})
\]

Input Parameters

\begin{itemize}
  \item \textit{parent} — The widget ID of the parent widget.
  \item \textit{label} — (optional) A string containing the label for the input field. If the \textit{label} parameter is either not defined or is defined as a null string, the function looks for the dialog label in a resource specification (see Discussion).
  \item \textit{OKCallback} — A string containing the name of the callback that is executed when the OK button is selected.
  \item \textit{CancelCallback} — A string containing the name of the callback that is executed when the Cancel button is selected.
  \item \textit{HelpCallback} — A string containing the name of the callback that is executed when the Help button is selected.
\end{itemize}

Returned Value

\[
wid — The ID of the dialog widget.
\]
Keywords

*Block* — If this keyword is present and nonzero, the dialog box is blocking (the default).

*Cols* — Specifies the number of columns in the text input field.

*Help* — Use this keyword to specify a help topic when the *HelpCallback* parameter is not specified. This keyword can specify a two-element or one-element string array. If it is a two-element array, the first element is the name of a help topic and the second is the filename of the help file containing the topic. A one-element array specifies only the name of a help topic. In this case the default help file is used. For information on the location of the default help file, refer to the HELP command (in the _PV-WAVE Reference_). The specified help topic is displayed in the online help viewer when the Help button is pressed.

*Name* — A string specifying the name of the PromptDialog widget. The keyword *Name* can be used in place of the *label* parameter, although *label* (if other than a null string) will take precedence if both are given. (Default: *dialog*.)

*Nonblock* — If this keyword is present an nonzero, the dialog box is not blocking. (This keyword has no effect under Microsoft Windows.)

*Text* — Specifies a string containing the initial text in the text input field.

*Title* — Specifies a string containing the dialog box title.

Color/Font Keywords

*Background* — Specifies the background color name.

*Font* — Specifies the name of the font used for text.

*Foreground* — Specifies the foreground color name.

Get/Set Value

*getvalue* — Gets a string containing the text entered in the text input field.

*setvalue* — Sets a string in the text input field.

Callback Parameters

Any dialog box callback procedure must have the following two parameters:

*wid* — Command widget ID.

*text* — Text input field widget ID.
Discussion

A dialog box is a *popup* window. This means that it cannot be the child of the top-level shell or the layout widget. Usually, a dialog box widget is activated by a push-button or menu button, as in the example below.

Part of the dialog label resource can be specified using the *Name* keyword, otherwise the default is the *.*.dialog.selectionLabelString resource (where selectionLabelString is the attribute).

**TIP** The *label* parameter provides a method for “hard-coding” the input field label in the application. For greater flexibility, create your resource file using a text editor, and load the resource containing the dialog label string using WtResource. The *Name* keyword can then be used in the WwDialog calling sequence to specify the dialog widget name in the resource specification.

Example 1

This example creates a button labeled **Dialog Box**. When the user selects this button, a dialog box appears. When the user enters text in the dialog box and presses <Return>, DialogOK is executed. When the user cancels the dialog box, the second callback routine, DialogCancel, is executed.

Enter the callback procedure into a file, and compile the procedure with the .RUN command. Then, enter the widget commands at the WAVE> prompt. To dismiss both widgets, select the appropriate function (such as Close) from the window manager menu of the **Dialog Box** button (the parent widget).

**Callback Procedures**

```
PRO DbuttonCB, wid, data
    select=WwDialog(wid,'Type something:',$
        'DialogOK','DialogCancel', Title='Type')
END

PRO DialogOK, wid, text
    PRINT,'Dialog OK'
    value = WwGetValue(text)
    PRINT, value
END

PRO DialogCancel, wid, data
    PRINT,'Dialog Cancel'
```
**Widget Commands**

```plaintext
top=WwInit('ww_ex23', 'Examples', layout)
button=WwButtonBox(layout, 'Dialog Box', 'DbuttonCB')
status=WwSetValue(top, /Display)
WwLoop
```

**Example 2**

A typical resource specification for the dialog prompt used in WwDialog is:

```plaintext
myapp.dialog.selectionLabelString: Enter some text:
```

**See Also**

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, *Using WAVE Widgets*.

For additional information on the color and font keywords, see *Setting Colors and Fonts* on page 201.

For information on Get and Set values, see *Setting and Getting Widget Values* on page 206.

---

**WwDrawing Function**

Creates a drawing area, which allows users to display graphics generated by PV-WAVE.

**Usage**

```plaintext
wid = WwDrawing(parent, windowid, drawCallback, wsize, dsize)
```

**Input Parameters**

- **parent** — The widget ID of the parent widget.
- **windowid** — The window ID of the PV-WAVE graphics window. When the window index is undefined or between zero and 31, the first free window index (ascending) is used and returned as `windowid`. (For information on window IDs, see the WINDOW procedure.)
**drawCallback** — A string containing the name of the PV-WAVE callback routine that is executed when the drawing area is exposed to display the graphics.

**NOTE** Avoid placing any calls directly in the `drawCallback` procedure that could create a window, such as a dialog box, in front of the drawing area widget. The reason for this is related to the way in which the X server handles backing store — the server’s method of repainting windows that are hidden (either entirely or partially) and then re-exposed. If your X server does not support backing store, or if it is running out of memory, it calls on your application to repaint windows. If the `drawCallback` procedure creates a graphics window and then places another window, such as a dialog box in front of the graphics window, when you dismiss the dialog box the X server calls the `drawCallback` procedure to repaint the drawing area, resulting in an infinite loop.

**wsize** — A vector containing two long integers that represent the width and height of the drawing area window. The window size is limited to 32767 pixels on a side, or the available memory for the pixmap, whichever is less.

**dsize** — A vector containing two long integers that represent the width and height of the image to be displayed in the drawing area. If this is larger than `wsize`, you can use the scroll bars to move the image around in the display window.

**Returned Value**

**wid** — The ID of the drawing area widget.

**Keywords**

**Area** — Returns the drawing area widget’s ID.

**Border** — Specifies the width of the borders in pixels for the parent widget and the child widgets. The default is 0.

**Layout_name** — Specifies the name of the Form or ScrolledWindow container widget used to hold the drawing area widget. The `Layout_name` specified is the top-level widget layout name used as part of the resource specification. (Default: `drawindow`.)

**Name** — A string containing the name of the DrawingArea widget. The `Name` specified is the widget name used as part of the resource specification. (Default: `draw`.)

**NoMeta** — (Windows only) Turns metafiles off for the window that contains the drawing area. Use this keyword when running animations or displaying images.
A metafile is an internal, vector-based record of all the graphics commands sent to a window. By default, a metafile is kept for each window to speed the redrawing of the window when it is resized. The metafile is also used when printing to avoid resolution problems that occur when printing a bitmap image.

**Noscroll** — If present and nonzero, the drawing area does not use scroll bars. The drawing area window is created the same size as the drawing area. In other words, the value of \textit{dsize} equals \textit{wsize}.

**Position** — If the drawing box widget is to be placed in a bulletin board layout, use this keyword to specify the \textit{x}, \textit{y} coordinates of the drawing box widget within the bulletin board.

**Color Keywords**

**Background** — Specifies the background color name.

**Foreground** — Specifies the foreground color name.

**Attachment Keywords**

**Bottom** — If a widget ID is specified (for example, \textit{Bottom}=\texttt{wid}), then the bottom of the drawing box widget is attached to the top of the specified widget. If no widget ID is specified (for example, \textit{/Bottom}), then the bottom of the drawing box widget is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, \textit{Left}=\texttt{wid}), then the left side of the drawing box widget is attached to the right side of the specified widget. If no widget ID is specified (for example, \textit{/Left}), then the left side of the drawing box widget is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, \textit{Right}=\texttt{wid}), then the right side of the drawing box widget is attached to the left side of the specified widget. If no widget ID is specified (for example, \textit{/Right}), then the right side of the drawing box widget is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, \textit{Top}=\texttt{wid}), then the top of the drawing box widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, \textit{/Top}), then the top of the drawing box widget is attached to the top of the parent widget.

**Get/Set Value**

**getvalue** — Gets the window ID of the drawing area widget.
**setvalue** — Sets a two-element vector containing the width and height of the drawing area.

**Callback Parameters**

Any drawing area widget callback procedure must have the following two parameters:

- **wid** — Drawing area widget ID.
- **index** — PV-WAVE window index.

**Discussion**

The window size is limited to 32767 pixels on a side, or the available memory for the pixmap, whichever is less.

**Example**

This example creates a widget that displays an image. Whenever the drawing area widget is displayed, the callback is executed. In this case, the callback opens and reads an image file.

Enter the callback procedure into a file, and compile the procedure with the .RUN command. Then, enter the widget commands at the **WAVE>** prompt. To dismiss the drawing area widget, select the appropriate function (such as **Close**) from the window manager menu.

**Callback Procedure**

```plaintext
PRO DrawCB, wid, data
    COMMON draw, img
    PRINT, 'Draw'
    TV, img
END
```

**Widget Commands**

```plaintext
top=WwInit('ww_ex24', 'Examples', layout)
COMMON draw, img
LOADCT, 5, /SILENT
img = BYTARR(512,512)
OPENR, 1, !Data_Dir + 'head.img'
```
WwFileSelection Function

Creates a file selection widget, which lets the user display the contents of directories and select files.

Usage

\[ wid = \text{WwFileSelection}(\text{parent}, \text{OKCallback}, \text{CancelCallback}[, \text{HelpCallback}]) \]

Input Parameters

- `parent` — The widget ID of the parent widget.
- `OKCallback` — A string containing the name of the callback that is executed when the OK button is selected.
- `CancelCallback` — A string containing the name of the callback procedure that is called when the file selection widget is dismissed.
- `HelpCallback` — (optional) A string containing the name of the callback routine that is called when the Help button is selected. The Help keyword can be used instead of this parameter.
Returned Value

\textit{wid} — The file selection widget ID.

Keywords

\textit{Block} — Creates a blocking file selection window (the default).

\textit{Dir} — Specifies a string containing the directory path.

\textit{File} — Specifies a string containing the default file selection.

\textit{Help} — Use this keyword to specify a help topic when the \textit{HelpCallback} parameter is not specified. This keyword can specify a two-element or one-element string array. If it is a two-element array, the first element is the name of a help topic and the second is the filename of the help file containing the topic. A one-element array specifies only the name of a help topic. In this case the default help file is used. For information on the location of the default help file, refer to the HELP command (in the \textit{PV-WAVE Reference}). The specified help topic is displayed in the online help viewer when the \textit{Help} button is pressed.

\textit{Name} — A string containing the name of the file selection box widget. The \textit{Name} specified is the widget name used as part of the resource specification. (Default: \textit{file}.)

\textit{NonBlock} — Creates a nonblocking file selection window. (This keyword has no effect under Microsoft Windows.)

\textit{Pattern} — Specifies the search pattern used in combination with the directory in determining files to be displayed.

\textit{Position} — Specifies the position of the upper-left corner of the file selection window on the screen in pixels.

\textit{Shell\_name} — Specifies the name of the TopLevelShell container widget used to hold the file selection box widget. The \textit{Shell\_name} specified is the top-level widget shell name used as part of the resource specification. (Default: \textit{fileshell}.)

\textit{Title} — Specifies a string containing the file selection widget’s title.

Color/Font Keywords

\textit{Background} — Specifies the background color name.

\textit{Font} — Specifies the name of the font used for text.

\textit{Foreground} — Specifies the foreground color name.
Get/Set Value

getvalue — Gets the selected file specification.

setvalue — Sets a three-element array of strings:

0 Determines the files and directories displayed in the directory list. For example, /usr/home/mydir/*.

1 Directory: specifies the base directory.

2 Pattern: specifies the search pattern to be used to select files.

Callback Parameters

Any file selection widget callback procedure must have the following two parameters:

wid — File selection widget ID.

shell — The ID of the top-level shell.

Discussion

A file selection widget is a pop-up window. This means that it cannot be the child of the top-level shell or the layout widget. Usually, a file selection widget is activated by a pushbutton or menu button, as in the example below.

Example

This example creates a button labeled File Selection. When the user selects this button, a file selection widget appears. When the user selects a file, the callback is executed.

Enter the callback procedure into a file, and compile the procedure with the .RUN command. Then, enter the widget commands at the WAVE> prompt. To dismiss the widget, select the appropriate function (such as Close) from the window manager menu of the File Selection button (the parent widget).

Callback Procedure

PRO FbuttonCB, wid, data
    file = WwFileSelection(wid,'FileOK', 'FileCancel', $ Title='Search')
END
PRO FileOK, wid, shell
    value = WwGetValue(wid)
    PRINT, value
    status = WwSetValue(shell, /Close)
END

PRO FileCancel, wid, shell
    PRINT, 'File Cancel'
END

**Widget Commands**

top = WwInit('ww_ex25', 'Examples', layout)
button = WwButtonBox(layout, 'File Tool', 'FbuttonCB')
status = WwSetValue(top, /Display)
WwLoop

**See Also**

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, *Using WAVE Widgets*.

For additional information on the color and font keywords, see *Setting Colors and Fonts* on page 201.

For information on Get and Set values, see *Setting and Getting Widget Values* on page 206.

---

**WwGenericDialog Function**

Creates a generic dialog box that can be filled with custom widgets.

**Usage**

(widget = WwGenericDialog(parent, layout [, labels] [, callback]))

**Input Parameters**

*parent* — The parent widget’s ID.
labels — (optional) A string or an array of strings containing the text that is to appear on the buttons in the Action area of the dialog box. This parameter determines the number of buttons in the Action area, and the text for the buttons.

callback — (optional) A string containing the name of the callback routine.

Output Parameters

layout — The widget ID of the layout widget (i.e., the widget returned by WwLayout). The layout widget must be filled in by the calling program.

Keywords

Block — If nonzero, the dialog box is blocking.

Board — If nonzero, a “bulletin board” layout is created.

Buttons — If specified, returns the WwButtonBox button widget IDs. For more information, see the WwButtonBox description.

Dialog_Name — A string containing the name of the generic dialog box.

Dismiss — An array of integers indicating which buttons in the Action area close the generic dialog box.

1 Indicates the button closes the dialog box.

0 Indicates the button doesn’t close the generic dialog box.

Form — If nonzero, a form layout is created.

Help — A string containing the name of an online Help Topic. If Help is specified, the right-most (last) Action area button is the Help button, regardless of its name or label. The Help keyword can also be specified as a two-element string array, whose first element is the help topic and whose second element is the Help file name.

Horizontal — If nonzero, child widgets are aligned horizontally within the layout widget. This keyword is only used for row/column layouts.

Layout_Name — An array of strings containing the names of the top-level and Action area widgets.

Name — An array of strings containing the names of the Action area buttons. If the number of elements in Name is less than the number of buttons, or if the labels
parameter was not specified, the remaining buttons are assigned default names of the form button_n, where n is the button index.

**NoDestroy** — If nonzero, the dialog box widget is hidden instead of destroyed when the **Cancel** button is selected.

**NonBlock** — If nonzero, the dialog box is not blocking. (This keyword has no effect under Microsoft Windows.)

**Scroll** — If nonzero, places scroll bars on the layout widget. If this keyword is specified, you must also provide the following parameters:

\[ w \] — Sets the width of the scrolled window.

\[ h \] — Sets the height of the scrolled window.

**Spacing** — Specifies the amount of space in pixels between child widgets in the layout. This keyword is only used for row/column layouts. (Default: 0)

**Title** — A string containing the title to appear on the border of the dialog box.

**Vertical** — If nonzero, child widgets are aligned vertically within the parent widget. This keyword is only used for row/column layouts.

**Color/Font Keywords**

**Background** — Specifies the background color name.

**Font** — Specifies the name of the font used for text.

**Foreground** — Specifies the foreground color name.

**Callback Parameters and Returned Value**

A generic dialog box callback must be a function containing the following two parameters:

\[ wid \] — The widget ID of a button in the Action area.

\[ index \] — The index of the Action area button pressed (1 – n).

**Returned Values**

0 Indicates the dialog box behaves according to the **Dismiss** keyword.
Discussion

The generic dialog box contains three major components:

- **Action area** — Contains a set of horizontal buttons at the bottom of the dialog box.
- **Separator** — Placed above the Action area.
- **Layout area** — An area that holds custom widgets. This area’s widget ID is returned by the output parameter `layout`.

Example

This example creates a dialog box containing some simple slider controls. Call-backs perform some simple actions whenever a slider is moved by the user.

Enter the following code into a file, and compile it with the .RUN command. To run the example, enter `WWGENERICDIALOG_TEST2` at the WAVE> prompt. Click the Create button in the main window to display the dialog box.

Button callback for the dialog created with WwGenericDialog.

This function always returns 0. If a 1 was returned instead of 0, the dialog would not be destroyed even if the Dismiss flag was set for the button pressed. This allows the callback to pop up an alert and avoid destroying the dialog if an error occurred.

```wavedoc
FUNCTION SimpleDialogCB, wid, data
    PRINT, 'SimpleDialogCB', wid, data
    RETURN, 0
END
```

Now, create the slider callback for the sliders in the dialog created with the WwGenericDialog function. This callback prints out the slider widget ID and the new value.

```wavedoc
PRO Slidercallback, wid, data
    value = WwGetValue (wid)
    PRINT, 'slidercallback', wid, data, value
END
```
The next callback creates the dialog using the WwGenericDialog function. The dialog created contains three sliders (Red, Green, and Blue), and three buttons (OK, Apply, and Cancel).

PRO wwgenericdialog_button_cb, wid, which
   IF which EQ 2 THEN BEGIN
   topshell = WwGetValue (wid, /Userdata)
   status = WwSetValue (topshell, /Close)
   RETURN
   ENDIF
   dialog_wid = WwGenericDialog (wid, layout, $
   ['OK', 'Apply', 'Cancel', 'Help'], $
   'SimpleDialogCB', $
   /Form, $
   Dismiss = [1, 0, 1], $
   Title = 'Simple Dialog', $
   Help = 'WwGenericDialog')
   ; Create the generic dialog.
controls = WwControlsBox(layout, $
   ['Red', 'Green', 'Blue'], $
   [0, 255, 0, 255, 0, 255], $
   'SliderCallback', $
   /Left, /Right, /Top, /Bottom, $
   /Vertical)
   ; Fill in the dialog layout.
status = WwSetValue (dialog_wid, /Show)
   ; Manage the dialog.
END

This is the destroy callback for the top-level shell.
PRO wwgenericdialog_destroy_cb, wid, data
   PRINT, 'wwgenericdialog_destroy_cb', wid, data
END

This is the main entry point for the test program. It creates a window with a button that creates a new dialog with the WwGenericDialog function when the button is pressed.

PRO Wwgenericdialog_test2
   topshell = WwInit ('wwgenericdialog_test2', $
   'Appl', workarea, $
   'Wwgenericdialog_destroy_cb', $
   background = background, $
   foreground = foreground, $
WwGetButton Function

Obtains the index of a pressed or released button passed as an event structure by a WAVE Widgets event handler.

Usage

\[
button = \text{WwGetButton}(\text{event})
\]

Input Parameter

\textit{event} — The event handle received by the event handler.

Returned Value

\textit{button} — The index of the button that was pressed or released. If the function call fails, the value –1L is returned.
**Keyword**

*State* — Returns a string array containing the state of all the buttons just before the event.

**Discussion**

The following are examples of strings found in the *State* array: ‘Button1’, ‘Button2’, ..., ‘ButtonN’ (indicating which button was pressed); and ‘Shift’, ‘Control’, ‘Lock’, ‘Mod1’, ..., (button modifiers).

**Example**

For an example using WwGetButton, see the example for the WwGetPosition function.

**See Also**

WwGetKey, WwGetPosition

---

**WwGetKey Function**

Obtains the ASCII value of a pressed or released key passed as an event structure by a WAVE Widgets event handler.

**Usage**

\[
key = \text{WwGetKey}(event)
\]

**Input Parameters**

*event* — The event handle received by the event handler.

**Returned Value**

*key* — A string containing the ASCII character of the released key. If the function call fails, an empty string is returned.
Keywords

Keysym — Returns a value associated with the key pressed.

UNIX USERS For Motif, Keysym returns a long value (XKeysym) associated with the key pressed. Keysym is an integer value unique to a particular key on the keyboard. This value can be used to identify function key presses.

Windows USERS For Windows, Keysym returns the long value of the virtual key associated with the key pressed (the result of the VkKeyScan procedure). For a list of virtual key codes and the keys to which they map, see Appendix E, Virtual Keys or refer to the Win32 Programmer’s Reference.

State — If specified, returns a string array containing the state of all the modifier keys just before the event.

Discussion

The following are examples of modifier key names found in the State array: ‘Shift’, ‘Control’, ‘Lock’, ‘Mod1’, ..., (indicating which key was pressed); and ‘Button1’, ‘Button2’, ..., ‘ButtonN’ (key modifiers).

The Keysym keyword is used in situations where the pressed key must be identified. The include file <X11/keysymdef.h> contains a complete Keysym listing.

Example

For an example using WwGetKey, see the example for the WwGetPosition function.

See Also

WwGetButton  WwGetPosition
WwGetPosition Function

Obtains the coordinates of a selected point inside a widget. The selected point coordinates are passed in an event structure by a WAVE Widgets event handler.

Usage

\[ point = \text{WwGetPosition}(event) \]

Input Parameters

\textit{event} — The event handle received by the event handler.

Returned Value

\textit{point} — A two-element array containing the position obtained from the \textit{event} parameter. If WwGetPosition fails, \([-1, -1]\) is returned.

Keywords

None.

Example

This example demonstrates the use of a number of WAVE Widgets in a simple application. The WAVE Widgets used in this example include: WwHandler, WwGetPosition, WwGetKey, WwSetCursor, and WwGetButton.

Two event handler procedures, DrawHandler and DrawKeyHandler, are registered with a drawing area widget:

- The DrawHandler procedure is called when a mouse button is pressed while the pointer is in the drawing area. DrawHandler prints the position of the pointer when the mouse button is pressed, which button was pressed, and the state of the modifier keys using the WwGetPosition and WwGetButton functions. Finally, DrawHandler removes itself from the drawing area widget.

- The DrawKeyHandler procedure is called when a key is pressed while the pointer is in the drawing area. DrawKeyHandler prints the ASCII character of the key, the state of the modifier keys, and the ASCII value of the key using the WwGetKey function.
If the key pressed is \textless w\textgreater, the drawing area cursor changes to the “wait cursor” (WwSetCursor). If the key pressed is \textless n\textgreater the drawing area cursor changes to the “default” cursor (WwSetCursor).

To run this example, enter the callback procedures into a file, and compile the procedure with the .RUN command. Then, enter the widget commands at the \texttt{WAVE}> prompt. To dismiss the drawing area widget, select the appropriate function (such as \texttt{Close}) from the window manager menu.

**Callback Procedures**

PRO DrawHandler, wid, shell, event

; This event handler procedure performs an action when a button is pressed.
COMMON draw, top, img
PRINT, ‘Position: ’, WwGetPosition(event)
; Print the position of the button press.
PRINT, WwGetButton(event, State = state)
; Print the button.
PRINT, ‘State: ’, state
; Print the modifier keys.
\texttt{h=WwHandler(wid, ‘DrawHandler’, /Remove)}
; Remove the handler.
END

PRO DrawKeyHandler, wid, shell, event

; This event handler procedure performs an action when a key is pressed.
COMMON draw, top, img
key = WwGetKey(event, State = state, Keysym = keysym)
; Print the ASCII value of the key, state of the modifier keys, and key symbol.
\texttt{IF key EQ ‘w’ THEN status = WwSetCursor(wid, /Wait)}
\texttt{IF key EQ ‘n’ THEN status = WwSetCursor(wid, /Default)}
; If the key is ‘w’, set the wait cursor; if the key is ‘n’ set the default cursor.
END

PRO DrawCB, wid, data

; A callback procedure to display an image.
COMMON draw, top, img
TV, img
END
Widget Commands

```plaintext
COMMON draw, top, img

top=WwInit('ww_ex100', 'Examples', layout)
LOADCT, 5, /SILENT
img=BYTARR(512,512)
OPENR,1, !Data_Dir + 'head.img'
READU,1,img
CLOSE, 1

draw=WwDrawing(layout, 1,'DrawCB', [256,256], [512,512], $
   Area = darea)
status = WwHandler(darea, 'DrawHandler', 'ButtonPressMask')
status = WwHandler(darea, 'DrawKeyHandler', 'KeyPressMask')
status=WwSetValue(top, /Display)
WwLoop
```

See Also

WwGetButton, WwGetKey

---

**WwGetValue Function**

Returns a specific value for a given widget.

**Usage**

```
value = WwGetValue(widget)
```

**Input Parameters**

- `widget` — The widget for which you want the value.

**Returned Value**

- `value` — The value returned from the `widget`. 
Keywords

**Children** — If nonzero, returns the widget IDs of the children of the widget specified by the *widget* parameter.

**Class** — If nonzero, returns the widget class of the widget specified by the *widget* parameter.

**Destroyed** — Returns 1 if the given widget is being destroyed; otherwise returns 0.

**Exists** — If nonzero, returns 1 if widget exists, 0 if not.

**Parent** — If nonzero, returns the widget ID of the parent of the widget specified by the *widget* parameter.

**Position** — Returns the position of the widget as a 2-element array: \([x, y]\), where \(x\) and \(y\) are a number of pixels.

**Sensitive** — Returns 1 if the widget is sensitive, or 0 if the widget is not sensitive.

**Shown** — Returns 1 if the widget is shown, or 0 if the widget is not visible.

**Size** — Returns the width and height of the widget as a 2-element array: \([w, h]\), where \(w\) and \(h\) are a number of pixels.

**NOTE** On shell widgets, the result of this keyword is always 1. This is because the Xt Intrinsics functions XtPopdown() and XtPopup() are always used to hide or show shell widgets, and it is not possible to determine whether a shell has been popped down or not.

**Userdata** — Returns the value of the *Userdata* variable that was previously stored with the WwSetValue function.

Discussion

See the *Get Value* section under each WAVE Widget function description to find out what WwGetValue returns by default for each function. For example, WwGetValue called with the ID of a list widget returns a string array containing the selected items in the list.

Example

The following example demonstrates WwGetValue with the WwCommand function. WwGetValue returns a string array containing the commands entered in the Command window. The callback routine *CommandOK* prints the value returned by WwGetValue.
WwGetValue whenever the user enters a command in the Command window and presses <Return>.

Enter the callback procedures into a file, and compile the procedure with the .RUN command. Then, enter the widget commands at the WAVE> prompt. To dismiss the widget, select the appropriate function (such as Close) from the window manager menu of the command window.

**Callback Procedures**

```plaintext
PRO CbuttonCB, wid, data
    command = WwCommand(wid, 'CommandOK', $ 'CommandDone', Position=[300,300], $ Title='Command Entry Window')
END

PRO CommandOK, wid, shell
    value = WwGetValue(wid)
    print, value
END

PRO CommandDone, wid, shell
    status = WwSetValue(shell, /Close)
END
```

**Widget Commands**

```plaintext
top=WwInit('ww_ex26', 'Examples', layout)
button=WwButtonBox(layout, 'Command', 'CbuttonCB')
status=WwSetValue(top, /Display)
WwLoop
```

**See Also**

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, *Using WAVE Widgets*. 
**WwHandler Function**

Adds or removes a WAVE Widgets event handler from a widget.

**Usage**

\[status = \text{WwHandler}(\text{wid}, \text{handler}[, \text{mask}[, \text{userdata}]]))\]

**Input Parameters**

- **wid** — The widget ID that the event handler is to be removed from or added to.
- **handler** — The name of the event handler procedure.
- **mask** — (optional) The mask for the event. This parameter must be specified if the Add keyword is used.
- **userdata** — (optional) Stores the value of the specified variable.

**Returned Value**

**status** — A value indicating success or failure of the addition or removal of the event handler.

- 1 Indicates success.
- 0 Indicates failure.

**Keywords**

- **Add** — Add an event handler for the event specified by the mask parameter. The mask input parameter must be specified when the Add keyword is used.
- **Remove** — Remove the specified event handler.

**Event Handler Parameters**

The following parameters are required for the event handler procedure:

- **wid** — The ID of the widget from which the event handler is called.
- **userdata** — Stores the value of the specified variable.
- **event** — An opaque event handle to be used in subsequent calls to request event data.
Discussion

The `mask` parameter can be specified as a value (such as, `ButtonPressMask`), if you include the Standard Library file `wtxlib.pro` using the `@` (at) command: `@wtxlib`.

One way to improve the portability of your application is to specify `mask` by name, instead of value. An example of specifying by name is: ‘`ButtonPressMask`’.

Example

For an example of `WwHandler`, see the example for the `WwGetPosition` function.

See Also

`WtAddHandler`, `WwMultiClickHandler`

---

**WwInit Function**

Initializes the WAVE Widgets environment, opens the display, creates the first top-level shell, and creates a layout widget.

**Usage**

```plaintext
topshell = WwInit(app_name, appclass_name, workarea [, destroyCallback])
```

**Input Parameters**

- `app_name` — A string containing the name of the application. This name can be referenced in a resource file.
- `appclass_name` — A string containing the application class name, which can be the name of a resource file.
- `destroyCallback` — (optional) A string containing the name of the callback that is executed when the top-level shell is destroyed.

**Output Parameters**

- `workarea` — The widget ID of the layout widget that is created inside of the top-level shell.
Returned Value

*topshell* — The widget ID of the top-level application shell.

Keywords

*Board* — If present and nonzero, a bulletin board layout is created inside the top-level shell.

*Border* — Specifies the width in pixels of the borders for the layout widget and its child widgets. Default is 0.

*Colors* — (Motif only) The maximum number of color table indices to be used. Otherwise, PV-WAVE uses all of the available color indices.

*ConfirmClose* — A string containing the name of the procedure called when the user selects the *Close* or *Quit* menu button from the window manager menu.

*Form* — If present and nonzero, a form layout is created inside the top-level shell.

*Height* — An integer specifying the height (in pixels) of the top level widget.

*Horizontal* — If present and nonzero, orients child widgets horizontally within the layout widget (the default). Used with row/column layouts only. For more information on layout widgets, see the *WwLayout* function.

*Layout_name* — Specifies the name of the layout widget. The *Layout_name* specified is part of the resource specification. The default for *Layout_name* depends on other keywords specified in the WwInit usage.

If either *Board* or *Form* are specified, the *Layout_name* default is *layout*.

If either *Horizontal* or *Vertical* are specified, the default for *Layout_name* is *workarea*.

*Position* — Specifies the position of the upper-left corner of the main window on the screen.

*Resource* — Specifies a particular resource file to load into the resource manager database. The resources in the file can be used to set attributes of the top-level shell created by WwInit. The defaults are:

(UNIX)  `<wavedir>/bin/Wave.ad`

(OpenVMS)  `<wavedir>:[BIN.WAVE.AD]`

(Windows)  `<wavedir>\bin\Wave.ad`

Where `<wavedir>` is the main PV-WAVE directory.
Shell_name — Specifies the name of the TopLevelShell widget. The Shell_name specified is the top-level widget shell name used as part of the resource specification. (Default: application_name.)

Spacing — Specifies the amount of space between child widgets inside the layout. Used with row/column layouts only. Default is 0. For more information on layout widgets, see the WwLayout function.

Title — A string specifying a title for the shell.

UserData — A variable. If the ConfirmClose keyword is specified, the value of this variable is passed to the Close or Quit callback procedure.

Vertical — If present and nonzero, orients child widgets vertically within the layout widget. Used with row/column layouts only. For more information on layout widgets, see the WwLayout function.

Width — An integer specifying the width (in pixels) of the top level widget.

Color/Font Keywords

Background — Specifies the default background color name for an application.

Font — Specifies the name of the default font used for text in an application.

Foreground — Specifies the default foreground color name for an application.

Discussion

Call this routine before the first use of a WAVE Widgets routine.

The ConfirmClose keyword lets you control what happens when the user selects Close or Quit from the window manager menu. Normally, the window from which the menu item was selected is destroyed; however, you might want to display a confirmation dialog box or take another action instead of simply allowing the window to be destroyed. The callback procedure specified by ConfirmClose destroys the window when appropriate.

The ConfirmClose procedure you specify accepts two parameters: wid and user_data, where:

\[ wid = \text{The widget ID of the top-level shell of the application.} \]
\[ user\_data = \text{The variable specified via the User_Data keyword. If User_Data is not specified, 0 (zero) is passed to the ConfirmClose routine.} \]

If specified, your ConfirmClose routine must close the top-level shell of the application. An example of a simple ConfirmClose routine which just closes the shell is:
PRO MyConfirmClose, wid, user_data
  s = WwSetValue(wid, /Close)
END

If ConfirmClose is not specified, then the shell is simply closed.

Example
For examples showing the use of WwInit, see any of the WAVE Widgets widget-creation routines, such as WwButtonBox, WwCommand, WwControlsBox, WwList, and so on.

See Also
For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, Using WAVE Widgets.

For additional information on the color and font keywords, see Setting Colors and Fonts on page 201.

**WwLayout Function**

Creates a layout widget that is used to control the arrangement of other widgets.

**Usage**

```
layout = WwLayout(parent)
```

**Input Parameters**

`parent` — The widget ID of the parent widget.

**Returned Value**

`layout` — The widget ID of the layout widget.

**Keywords**

`Board` — If present and nonzero, the layout that is created is a “bulletin board”.

---

WwLayout Function 279
**Border** — Specifies the width of the borders in pixels for the parent widget and the child widgets. The default is 0.

**Form** — If present and nonzero, a form layout is created.

**Frame** — If nonzero, creates a frame widget with a layout child and returns the widget ID for the layout.

**Horizontal** — If present and nonzero, aligns child widgets horizontally within the layout widget (the default). This keyword is only used for row/column layouts.

**Layout_name** — Specifies the name of the layout widget used as part of the resource specification. *Layout_name* names the Form widget (when used with *Form*), the RowColumn widget (when used with *Horizontal* and/or *Vertical*), the ScrolledWindow widget (when used with *Scroll*), or the BulletinBoard widget (when used with *Board*). (Default: *layout*.)

**Name** — A string specifying the name of the frame widget created using the *Frame* keyword. The frame widget name specified is part of the resource specification. (Default: *frame*.)

**Position** — If the layout widget is to be placed in a bulletin board layout, use this keyword to specify the $x$, $y$ coordinates of the layout widget within the bulletin board.

**Scroll** — If present and nonzero, places scroll bars on the layout widget. Specify a width and height (e.g., *Scroll* = [w, h]) to set the width and height of the scrolled window.

**Spacing** — Specifies the amount of space in pixels between child widgets in the layout. The default is 0. This keyword is only used for row/column layouts.

**Vertical** — If present and nonzero, aligns child widgets vertically within the parent widget. This keyword is only used for row/column layouts.

**Color/Font Keywords**

**Background** — Specifies the background color name.

**Foreground** — Specifies the foreground color name.

**Attachment Keywords**

**Bottom** — If a widget ID is specified (for example, *Bottom=wid*), then the bottom of the layout widget is attached to the top of the specified widget. If no widget ID is specified (for example, */Bottom*), then the bottom of the layout widget is attached to the bottom of the parent widget.
Left — If a widget ID is specified (for example, \texttt{Left=\textit{wid}}), then the left side of the layout widget is attached to the right side of the specified widget. If no widget ID is specified (for example, \texttt{/Left}), then the left side of the layout widget is attached to the left side of the parent widget.

Right — If a widget ID is specified (for example, \texttt{Right=\textit{wid}}), then the right side of the layout widget is attached to the left side of the specified widget. If no widget ID is specified (for example, \texttt{/Right}), then the right side of the layout widget is attached to the right side of the parent widget.

Top — If a widget ID is specified (for example, \texttt{Top=\textit{wid}}), then the top of the layout widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, \texttt{/Top}), then the top of the layout widget is attached to the top of the parent widget.

Get/Set Value

\texttt{getvalue} — Gets the $x$, $y$ position of the contents of the scrolled window.

\texttt{setvalue} — Sets the $x$, $y$ position of the contents of the scrolled window.

Callback Parameters

None.

Discussion

The \texttt{Board}, \texttt{Form}, \texttt{Vertical}, \texttt{Horizontal}, and \texttt{Scroll} keywords are mutually exclusive: you can only use one of them at a time.

A layout widget is a container that holds other widgets. The layout widget provides different methods of arranging widgets, such as buttons, menus, sliders, and even other layout widgets, inside the “container.” The three layout types are:

- Row/column — Widgets are arranged in rows and/or columns (the default).
- Form — Widgets are “attached” to one another inside the layout. Specified with the \texttt{Form} keyword.
- Bulletin board — Widgets are positioned in the layout with $x$, $y$ coordinates. Specified with the \texttt{Board} keyword.

The layout widget can be the parent widget for each “child” widget it contains. If a border is not specified with the \texttt{Border} keyword, the layout widget itself is not visible to the user; only the widgets inside the layout are visible.
For more detailed information on layouts, see *Arranging Widgets in a Layout* on page 169.

**Example**

The following example creates a simple layout widget containing a button box and a radio box widget. Note that WwLayout does not use callbacks, as it primarily creates a “container” that holds other widgets.

Enter the callback procedures into a file, and compile them with the .RUN command. Then, enter the widget commands at the WAVE> prompt (or enter them in a command file and run them with the @ command). To dismiss the layout box, select the appropriate function (such as **Close**) from the window manager menu.

**Callback Procedures**

PRO RadioCB, wid, which

CASE which OF
    1: PRINT,’First Toggle Selected’
    2: PRINT,’Second Toggle Selected’
    3: PRINT,’Third Toggle Selected’
ENDCASE

value = WwGetValue(wid)
PRINT, value

END

PRO ButtonCB, wid, data

CASE data OF
    1: PRINT,’Quit Selected’
    2: PRINT,’Dialog Selected’
    3: PRINT,’Message Selected’
ENDCASE

END

**Widget Commands**

top=WwInit(‘ww_ex27’, ‘Examples’, layout,$
    /Vertical, Spacing=30, Border=10)
blabels = [‘Quit’, ‘Dialog’, ‘Message’]
bbox = WwButtonBox(layout, blabels,’ButtonCB’,$
    /Horizontal, Spacing=20)
rlabels = [‘System’, ‘Owner’, ‘Group’]
rbox = WwRadioBox(layout, rlabels, 'RadioCB', $ /Vertical, Border=2, Spacing=20, $ Top=controls)
status = WwSetValue(top, /Display)
WwLoop

See Also

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, *Using WAVE Widgets*.

For additional information on the color and font keywords, see *Setting Colors and Fonts* on page 201.

For additional information on attachment keywords, see *Form Layout: Attachments* on page 170.

For information on Get and Set values, see *Setting and Getting Widget Values* on page 206.

### WwList Function

Creates a scrolling list widget.

**Usage**

\[
list = WwList(parent, [items,] selectedCallback, defaultCallback)
\]

**Input Parameters**

- **parent** — The widget ID of the parent widget.
- **items** — (optional) A string array containing the items to appear on the list. If the **items** parameter is undefined or is defined as an array of null strings, the function looks for item strings in a resource specification (see *Discussion*).
- **selectedCallback** — A string containing the name of the callback that is executed when an item is selected.
- **defaultCallback** — A string containing the name of the callback that is executed when a user double-clicks on an item.
Returned Value

list — The widget ID of the list widget.

Keywords

Browse — If specified and nonzero, the list uses the “browse” selection method. This method allows the user to select at most one item at a time. Whenever the user selects an item, the currently selected item is deselected. When the user presses the mouse selection button and drags the pointer over the list, the current selection moves along with the pointer.

Extended — If specified and nonzero, the list uses the “extended” selection method. This method allows the user to select multiple items at a time. Whenever the user selects an item, the currently selected item is deselected; however, when the user presses the mouse selection button and drags the pointer over the list, multiple items are selected. The selected items include all items between the item on which the mouse selection button was pressed and the item currently under the pointer.

HorzSb — If present and nonzero, a horizontal scroll bar appears if the list contents are wider than the list.

Multi — If present and nonzero, the list widget uses multiple selection mode. The default is single selection mode.

Name — A string specifying the name of the List widget or an array of strings specifying the List widget name and the item names if the items parameter is not defined. This is part of the resource specification. The Name keyword can be used in place of the items parameter, although items will take precedence if both are given. (Default: list.)

Position — If the scrolling list widget is to be placed in a bulletin board layout, use this keyword to specify the x, y coordinates of the scrolling list widget within the bulletin board.

Selected — Specifies the string, or string array, of items to be initially selected in the list.

Visible — Specifies the number of items that are visible in the list widget.

Color/Font Keywords

Background — Specifies the background color name.

Font — Specifies the name of the font used for text.
**Foreground** — Specifies the foreground color name.

**Attachment Keywords**

**Bottom** — If a widget ID is specified (for example, `Bottom=wid`), then the bottom of the scrolling list widget is attached to the top of the specified widget. If no widget ID is specified (for example, `/Bottom`), then the bottom of the scrolling list widget is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, `Left=wid`), then the left side of the scrolling list widget is attached to the right side of the specified widget. If no widget ID is specified (for example, `/Left`), then the left side of the scrolling list widget is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, `Right=wid`), then the right side of the scrolling list widget is attached to the left side of the specified widget. If no widget ID is specified (for example, `/Right`), then the right side of the scrolling list widget is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, `Top=wid`), then the top of the scrolling list widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, `/Top`), then the top of the scrolling list widget is attached to the top of the parent widget.

**Get/Set Value**

`getvalue` — Returns an array of the selected items (string).

`setvalue` — Replaces current items with the new items (array of strings).

**Callback Parameters**

Any list widget callback procedure must have the following two parameters:

`wid` — List widget ID.

`parent` — Parent widget ID.

**Discussion**

A scrolling list allows users to select one or more items from a group of choices. Items are selected from the list with the mouse. An additional callback can be defined for the default action. This callback is executed when the user double-clicks on an item.
Part of the list item resource can be specified using the *Name* keyword, otherwise the default is the *\*.item_\#.itemString* resource (where *itemString* is the attribute).

**TIP** The *items* parameter provides a method for “hard-coding” the list items in the application. For greater flexibility, create your resource file using a text editor, and load the resource containing the list of items using WtResource. The *Name* keyword can then be used in the WwList calling sequence to specify the List widget name and items in the resource specification.

**Example 1**

This example creates a scrolling list containing the items defined in the string array *items*. When the user selects an item (clicks it with the mouse), the first callback *ListCB* is executed. If the user double-clicks on an item, the callback *DefaultCB* is executed.

Enter the callback procedures into a file, and compile them with the .RUN command. Then, enter the widget commands at the WAVE> prompt. To dismiss the list box, select the appropriate function (such as Close) from the window manager menu.

**Callback Procedures**

```plaintext
PRO ListCB, wid, data
    PRINT,'Item Selected'
    value = WwGetValue(wid)
    PRINT, value
END

PRO DefaultCB, wid, data
    PRINT,'Default Item Selected'
    value = WwGetValue(wid)
    PRINT, value
END
```

**Widget Commands**

```plaintext
top = WwInit('ww_ex28', 'Examples', layout)
items = ['Presidents Day','St.Patricks Day', $
    'Easter', 'Memorial Day', '4th of July', $
    'Labor Day', 'Halloween', 'Thanksgiving', $
    'Hanukkah', 'Christmas', 'New Years Eve']
```
datelist = WwList(layout, items, ‘ListCB’, ‘DefaultCB’, Visible=7, /Multi)
status = WwSetValue(top, /Display)
WwLoop

Example 2
A typical resource specification for the list items used in WwList is:
Name = ['mylist', 'spring', 'summer']
myapp.mylist.summer.itemString: Summer

See Also
WwListUtils
For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, Using WAVE Widgets.
For additional information on the color and font keywords, see Setting Colors and Fonts on page 201.
For additional information on attachment keywords, see Form Layout: Attachments on page 170.
For information on Get and Set values, see Setting and Getting Widget Values on page 206.

WwListUtils Function
Manages the contents of a list widget.

Usage
status = WwListUtils(wid [, param1[, param2]])

Input Parameters
wid — The widget ID of a scrolling list created with WwList.
param1 — (optional) This parameter depends on keyword use in the function calling sequence. See Keywords for more information.
**param2** — (optional) This parameter depends on keyword use in the function calling sequence. See **Keywords** for more information.

**Returned Value**

**status** — A value indicating the success or failure of the WwListUtils call.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Indicates success.</td>
</tr>
<tr>
<td>0</td>
<td>Indicates failure.</td>
</tr>
</tbody>
</table>

**Keywords**

**Add** — Add the items specified by **param1** to the list following the list position specified by **param2**. If this keyword is used, you must also specify both of the following parameters.

- **param1** — A string or array of strings containing the items to add to the list.
- **param2** — (long) Specifies the position in the list below which the items are added. (Default: end of the list)

**Delete** — Delete the specified items from the list. If the **All** modifier keyword is also specified, all items in the list are deleted. If **All** is not specified with **Delete**, you must specify the following parameter:

- **param1** — A string or array of strings containing the items to delete from the list.

**Deselect** — Deselect the specified items from the list. If the **All** modifier keyword is also specified, all items in the list are deselected. If **All** is not specified with **Deselect**, you must specify the following parameter:

- **param1** — A string or array of strings containing the items to deselect in the list.

**GetItems** — If nonzero, returns an array of strings in an output parameter **param1** containing all the items in the list. If the modifier keyword **Count** is specified, the number of items is returned rather than the items themselves. If the **Count** keyword is used, you must specify the following output parameter:

- **param1** — An array of strings, or a long integer (optional) representing the number of items in the list is returned.
**GetSelected** — If nonzero, returns an array of strings in an output parameter `param1` containing the selected items. If the modifier keyword *Count* is specified, the number of selected items is returned rather than the items themselves. If the *Count* keyword is used, you must specify the following parameter:

`param1` — An array of strings, or a long integer (optional) representing the number of selected items in the list is returned.

**Replace** — Replace the specified items in the list, starting at a specified list position. If this keyword is used, you must specify both of the following parameters.

`param1` — A string or array of strings containing the items to replace.

`param2` — (long) The position in the list to begin replacing items.

**Select** — Select the specified items in the list. If the *All* modifier keyword is also specified, all items in the list are selected. If *All* is not specified with *Select*, you must specify the following parameter:

`param1` — A string or array of strings containing the items to select in the list.

If the modifier keyword *Notify* is used with *Select*, the `selectedCallback` registered with the WwList routine is called.

**Modifier Keywords**

**NOTE** The following modifier keywords are only used in conjunction with the WwListUtils keywords listed.

**All** — If nonzero, all items in the list are selected, deselected, or deleted.

**Count** — If nonzero and the *GetSelected* or *GetItems* keywords are specified, the number of items in the list widget is returned.

**Notify** — If nonzero and the *Select* keyword is specified, calls a PV-WAVE callback routine for the selected items.

**Discussion**

WwListUtils facilitates portability of PV-WAVE applications between Microsoft Windows and X Windows environments. The WwListUtils function is a wrapper for the WtList function.
**WwLoop Procedure**

Handles the dispatching of events and calling of callbacks.

**Usage**

WwLoop

**Parameters**

None.

**Returned Value**

None.

**Keywords**

*Noblock* — If specified and nonzero, events are dispatched in the background, and WwLoop returns immediately to process PV-WAVE commands from the command line.

**Discussion**

WwLoop causes PV-WAVE to loop indefinitely, processing the events and dispatching callbacks. WwLoop is always the last WAVE Widgets command in a WAVE Widgets procedure.

**NOTE** For interapplication development using *cwavec()* or *cwavefor()* , the default behavior of WwLoop is to block, even if the *Noblock* keyword is set. To force nonblocking, set Noblock = 2 and then call WtProcessEvent periodically to service the event loop.

**See Also**

WtList, WwList
Example
For examples showing the use of WwLoop, see any of the WAVE Widgets widget-creation routines, such as WwButtonBox, WwCommand, WwControlsBox, WwList, and so on.

See Also
WtProcessEvent
For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, Using WAVE Widgets.

WwMainWindow Function
Creates a top-level window and a layout widget.

Usage
shell = WwMainWindow(parent, workarea, [destroyCallback])

Input Parameters
parent — The widget ID of the parent widget.
destroyCallback — A string containing the name of the callback that is executed when the main window is destroyed.

Output Parameters
workarea — The widget ID of the layout widget that is created inside of the top-level shell.

Returned Value
shell — The widget ID of the top-level shell.

Keywords
Board — If present and nonzero, a bulletin board layout is created inside the main window shell. See WwLayout for more information.
**Border** — Specifies the width in pixels of the borders for the layout widget and its child widgets. Default is 0.

**ConfirmClose** — A string containing the name of the procedure called when the user selects the Close or Quit menu button from the window manager menu.

**Form** — If present and nonzero, a form layout is created inside the shell. See WwLayout for more information.

**Horizontal** — If present and nonzero, aligns child widgets horizontally within the layout widget (the default). Used with row/column layouts only. For more information on layout widgets, see the WwLayout function.

**Layout_name** — Specifies the name of the layout widget which is part of the resource specification. The default depends on other keywords specified in the WwMainWindow usage.

If either Board or Form are specified, the Layout_name default is layout.

If either Horizontal or Vertical are specified, the default for Layout_name is workarea.

**Position** — Specifies the x, y position of the upper-left corner of the main window on the screen.

**Shell_name** — Specifies the name of the top-level layout widget, or TopLevelShell which is part of the resource specification. (Default: shell.)

**Spacing** — Specifies the amount of space in pixels between child widgets inside the layout. Used with row/column layouts only. Default is 0. For more information on layout widgets, see the WwLayout function.

**Title** — A string specifying a title for the shell.

**UserData** — A variable. If the ConfirmClose keyword is specified, the value of this variable is passed to the Close or Quit callback procedure.

**Vertical** — If present and nonzero, aligns child widgets vertically within the layout widget. Used with row/column layouts only. For more information on layout widgets, see the WwLayout function.

**Color/Font Keywords**

**Background** — Specifies the background color name.

**Font** — Specifies the name of the font used for text.

**Foreground** — Specifies the foreground color name.
Get/Set Value
Not supported.

Callback Parameters
Any main window callback procedure must have the following two parameters:
shell — Main window shell widget ID.
layout — Layout widget ID.

Discussion
The WwInit function creates one main window, and for many applications, this is sufficient. WwMainWindow gives you the ability to create additional main windows and layout widgets.

By default, the WwMainWindow creates a row/column layout widget. For more information on layout widgets, see WwLayout.

The ConfirmClose keyword lets you control what happens when the user selects Close or Quit from the window manager menu. Normally, the window from which the menu item was selected is destroyed; however, you might want to display a confirmation dialog box or take another action instead of simply allowing the window to be destroyed. The callback procedure specified by ConfirmClose destroys the window when appropriate.

The ConfirmClose procedure you specify accepts two parameters: wid and user_data, where:

wid = The widget ID of the top-level shell of the application.
user_data = The variable specified via the User_Data keyword. If User_Data is not specified, 0 (zero) is passed to the ConfirmClose routine.

If specified, your ConfirmClose routine must close the top-level shell of the application. An example of a simple ConfirmClose routine which just closes the shell is:

```
PRO MyConfirmClose, wid, user_data
    s = WwSetValue(wid, /Close)
END
```

If ConfirmClose is not specified, then the shell is simply closed.
Example

This example shows a call to the WwMainWindow function. No callback is specified.

\[
\text{shell} = \text{WwMainWindow}(\text{parent, form, destroyCB, }$
\quad \text{/Vertical, Title = 'Topics')}
\]

See Also

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, *Using WAVE Widgets*.

For additional information on the color and font keywords, see *Setting Colors and Fonts* on page 201.

---

**WwMenuBar Function**

Creates a menu bar.

**NOTE** WwMenuBar can only occur once per layout widget on Windows.

Usage

\[
\text{menubar} = \text{WwMenuBar}(\text{parent, items})
\]

Input Parameters

- **parent** — The widget ID of the parent widget.
- **items** — An unnamed structure containing the menu bar items. For more information, see *Creating and Handling Menus* on page 173.

Returned Value

- **menubar** — The widget ID of the menu bar widget.

Keywords

- **HelpJustify** — If nonzero, positions the last menu item (usually the Help menu) at the far right of the menu bar.
Layout_name — Specifies the name of the RowColumn widget which is part of the resource specification. (Default: menubar.)

Menus — Returns an array of menu pane widget IDs in the order in which the menus were created. A menu pane is a special menu widget that serves as a container for a menu item. Menu pane widget IDs can be used in the WwMenuItem function to add, modify, or delete menu items.

Position — (UNIX/OpenVMS only) If the menu bar widget is to be placed in a bulletin board layout, use this keyword to specify the x, y coordinates of the menu bar widget within the bulletin board.

Spacing — Specifies the amount of space in pixels between child widgets.

Color/Font Keywords

Background — Specifies the background color name.

Font — Specifies the name of the font used for text.

Foreground — Specifies the foreground color name.

Attachment Keywords

Bottom — If a widget ID is specified (for example, Bottom=wid), then the bottom of the menu bar widget is attached to the top of the specified widget. If no widget ID is specified (for example, /Bottom), then the bottom of the menu bar widget is attached to the bottom of the parent widget.

Left — If a widget ID is specified (for example, Left=wid), then the left side of the menu bar widget is attached to the right side of the specified widget. If no widget ID is specified (for example, /Left), then the left side of the menu bar widget is attached to the left side of the parent widget.

Right — If a widget ID is specified (for example, Right=wid), then the right side of the menu bar widget is attached to the left side of the specified widget. If no widget ID is specified (for example, /Right), then the right side of the menu bar widget is attached to the right side of the parent widget.

Top — If a widget ID is specified (for example, Top=wid), then the top of the menu bar widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, /Top), then the top of the menu bar widget is attached to the top of the parent widget.
Get/Set Value

getvalue — Gets the button label or icon pixmap ID for the selected menu button. Or, if the button is a toggle, determines if the button is selected or unselected (selected = 1, unselected = 0).

setvalue — Sets the button label or icon pixmap ID for the selected menu button. Or, if the button is a toggle, selects or unselects the toggle button (select = 1, unselect = 0).

Callback Parameters

Any menu bar callback procedure must have the following two parameters:

wid — Widget ID of the selected menu item.
index — Index of the selected menu item (1 – n).

Example

First, this example creates a menu bar with three menus: Fonts, Size, and Icons. The unnamed structure menus contains all of the information used to create the menus.

Enter the callback procedure into a file, and compile the procedure with the .RUN command. Then, enter the widget commands at the WAVE> prompt (or enter them in a command file and run them with the @ command). To dismiss the menu bar, select the appropriate function (such as Close) from the window manager menu.

Callback Procedure

PRO MenuCB, wid, index
    PRINT, 'Menu Item', index, 'selected.'
    value = WwGetValue(wid)
    PRINT, value
END

Widget Commands

top = WwInit('ww_ex29', 'Examples', layout)
menus={,callback:'MenuCB', $
    menubutton:'Fonts', $
    menu:{,callback:'MenuCB', $
        menubutton:'Adobe', $
        menu: {},callback:'MenuCB', $
        menu: {},callback:'MenuCB', $

toggle:'Normal', $
  toggle:'Bold', $
  toggle:'Italic'}, $
button:'Helvetica', $
button:'Courier'}, $
menubutton:'Size', $
menu:{,callback:'MenuCB', $
  button:'8', $
  button:'10', $
  button:'12'}, $
menubutton:'Icons', $
menu:{,callback:'MenuCB', title:'Help', $
icon:getenv('WAVE_DIR')+ $
  '/xres/wxbm_btn_help_search.',  $
icon:getenv('WAVE_DIR')+ $
  '/xres/wxbm_btn_help_toc.',  $
icon:getenv('WAVE_DIR')+ $
  '/xres/wxbm_btn_help_topics.', $
separator:1,  $
icon:getenv('WAVE_DIR')+ $
  '/xres/wxbm_btn_help_quit.'}}$
bar=WwMenuBar(layout, menus)
status=WwSetValue(top, /Display)
WwLoop

See Also

WwMenuItem, WwOptionMenu, WwPopupMenu

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, Using WAVE Widgets.

For additional information on the color and font keywords, see Setting Colors and Fonts on page 201.

For additional information on attachment keywords, see Form Layout: Attachments on page 170.

For information on Get and Set values, see Setting and Getting Widget Values on page 206.
**WwMenuItem Function**

Adds, modifies, or deletes specified menu items.

**Usage**

\[status = \text{WwMenuItem}(\text{parent}, \text{item}, \text{value}[, \text{callback}])\]

**Input Keywords**

- **parent** — The menu pane widget ID acquired using the *Menus* keyword from the WwMenuBar, WwOptionMenu, or WwPopupMenu functions. A menu pane is a special menu widget that serves as a container for a menu item.

- **item** — The index of the menu item; the index of the first menu item is one (1). Used for an Update or Delete operation.

- **value** — The value (string) of the menu item. Used for an Update or Add operation.

- **callback** — The callback procedure that is executed when the menu item is selected. Used for an Add operation.

**Returned Value**

- **status** — Returns one (1) if the function is successful, or zero (0) if the function is not successful.

**Keywords**

**Add** — Appends the specified *item* to the menu. The type of item is specified by additional keywords (*Button*, *Icon*, *Menu*, or *Toggle*). The *Name* keyword can also be used with *Add* to specify the resource specification for the particular type of widget.

- **Button** — A push button item is added. If *Name* is not specified, the default resource specification is *button___#*.

- **Icon** — A graphic (icon) button is added. If *Name* is not specified, the default resource specification is *icon___#*.

- **Menu** — A pull-right menu item is added. If *Name* is not specified, the default resource specification is *pane___#*.

- **Toggle** — A toggle (radio) button is added. If *Name* is not specified, the
default resource specification is `toggle__#`.

**NOTE** The default resource specification for each item type consists of the item name followed by two `_` (underscores) and an index number identifying the number of items in the menu.

**Update** — Modifies the value of the item specified by the index of `item`.

**Delete** — Deletes the item specified by `index`.

**Name** — A string specifying the name of the PushButton widget (if a string or icon label is given), otherwise `Name` specifies the name of the ToggleButton widget. The `Name` keyword is used as part of the resource specification that identifies items specified using the `Add` keyword. The defaults for `Name` are listed for each of the three different item types under the `Add` keyword description.

**Discussion**

WwMenuItem lets you dynamically update menus that have already been created. All menu items are placed inside a parent menu pane, and the widget ID of the appropriate menu pane can be acquired using the `Menus` keyword of the WwMenuBar, WwPopupMenu, or WwOptionMenu functions.

To update (`Update` keyword) or remove (`Delete` keyword) a menu item, use the appropriate menu item index; the index of the first menu item is one (1). To add a menu item to the bottom of the menu use the `Add` keyword.

**Example**

The following fragment shows how a callback might be written that uses WwMenuItem to modify the contents of a menu pane.

```pro
PRO ButtonCB, wid, index
  COMMON Menus, menupane, item_no, new_name
  CASE index OF
    1: BEGIN ; Add new item
      status = WwMenuItem(menupane, item_no, new_name, /Add)
    END
    2: BEGIN; Update last selected item
      status = WwMenuItem(menupane, item_no, new_name, $ /Update)
    END
    3: BEGIN ; Remove last selected item
```
status = WwMenuItem(menupane, item_no, /Delete)
END
ENDCASE
END

See Also
For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, Using WAVE Widgets.

WwMessage Function

Creates a blocking or nonblocking message box.

Usage

\[
\text{wid} = \text{WwMessage}(\text{parent}, [\text{label},] \text{OKCallback, CancelCallback, HelpCallback})
\]

Input Parameters

\text{parent} — The widget ID of the parent widget.
\text{label} — (optional) A string containing the message text of the Dialog widget. If the \text{label} parameter is not defined or is a null string, the function looks for the message label in a resource specification (see Discussion).
\text{OKCallback} — A string containing the name of the callback routine that is called when the \text{OK} button is selected.
\text{CancelCallback} — A string containing the name of the callback routine that is executed when the \text{Cancel} button is selected.
\text{HelpCallback} — A string containing the name of the callback that is executed when the \text{Help} button is selected.

Returned Value

\text{wid} — The message box widget ID.

Keywords

\text{Block} — If present and nonzero, creates a blocking message box (the default).
**Help** — Use this keyword to specify a help topic when the *HelpCallback* parameter is not specified. This keyword can specify a two-element or one-element string array. If it is a two-element array, the first element is the name of a help topic and the second is the filename of the help file containing the topic. A one-element array specifies only the name of a help topic. In this case the default help file is used. For information on the location of the default help file, refer to the HELP command (in the *PV-WAVE Reference*). The specified help topic is displayed in the online help viewer when the **Help** button is pressed.

**Info** — If present and nonzero, creates an information message box.

**Name** — A string specifying the name of the Dialog widget. The *Name* specified is the top-level widget name used in the resource specification. The keyword *Name* can be used in place of the *label* parameter, although *label* (if other than a null string) will take precedence if both are given. (Default: *message*.)

**Nonblock** — If present and nonzero, creates a nonblocking message box. (This keyword has no effect under Microsoft Windows.)

**Question** — If present and nonzero, creates a question message box.

**Title** — Specifies a string containing the message box title.

**Warning** — If present and nonzero, creates a warning message box.

**Working** — If present and nonzero, creates a working message box.

**Color/Font Keywords**

**Background** — Specifies the background color name.

**Font** — Specifies the name of the font used for message text.

**Foreground** — Specifies the foreground color name.

**Get/Set Value**

Not supported.

**Callback Parameters**

Any message box callback procedure must have the following two parameters:

**wid** — Message widget ID.

**shell** — Shell widget ID.
Discussion

A message window is a *popup* window. This means that it cannot be the child of the top-level shell or the layout widget. Usually, a message widget is activated by a pushbutton or menu button, as in the example below.

Part of the message text resource specification can be specified using the *Name* keyword, otherwise the default is the *message.messageString* resource (where *messageString* is the attribute).

**TIP**  The *label* parameter provides a method for “hard-coding” the message text in the application. For greater flexibility, create your resource file containing the message text using a text editor, and load the resource using WtResource. The *Name* keyword can then be used in the WwMessage calling sequence to specify the Dialog widget name in the resource specification.

Example 1

This example creates a button box with four buttons. If you are running under Motif, each button activates one of the four types of message windows: information, working, warning, or question. The callback *MessageOK* is executed when the user clicks on the *OK* or *Confirm* button. The callback *MessageCancel* is executed when the user clicks on the *Cancel* button.

Enter the callback procedures into a file, and compile them with the .RUN command. Then, enter the widget commands at the WAVE> prompt. To dismiss the widgets, select the appropriate function (such as *Close*) from the window manager menu of the menu bar.

**Callback Procedures**

```
PRO MbuttonCB, wid, data
    CASE data OF
      1: message=WwMessage(wid, $ 'This is a Test Message','MessageOK', $ 'MessageCancel',Title='Information')
      2: message=WwMessage(wid, $ 'This is a Test Message','MessageOK', $ 'MessageCancel', /Working, Title='Working')
      3: message=WwMessage(wid, $ 'This is a Test Message','MessageOK', $ 'MessageCancel', /Warning, $ Title='Warning')
```

4: message=WwMessage(wid, $ 
  'This is a Test Message','MessageOK', $ 
  'MessageCancel', /Question, Title='Question') 
ENDCASE 
END 
PRO MessageOK, wid, data 
  PRINT,'Message OK' 
END 
PRO MessageCancel, wid, data 
  PRINT,'Message Cancel' 
END 

**Widget Commands**

    top = WwInit('ww_ex30', 'Examples', layout) 
    button = WwButtonBox(layout, ['Information', $ 
        'Working', 'Warning', 'Question'], $ 
        'MbuttonCB') 
    status = WwSetValue(top, /Display) 
    WwLoop 

**Example 2**

A typical resource specification for dialog text used in WwMessage is:

    myapp.message.messageString: This is my message.

**See Also**

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, *Using WAVE Widgets*.

For additional information on the color and font keywords, see *Setting Colors and Fonts* on page 201.
WwMultiClickHandler Function

Adds or removes a multi-click event handler.

Usage

\[
status = \text{WwMultiClickHandler}(\text{wid}, \text{handler} [, \text{userdata}])
\]

Input Parameters

\text{wid} — The widget ID of the widget to which the multi-click handler is added or from which it is removed.

\text{handler} — The name of the multi-click handler procedure.

\text{userdata} — (optional) Stores the value of the specified variable.

Returned Value

\text{status} — A value indicating the success or failure of the addition or removal of the multi-click event handler.

\[
\begin{align*}
1 & \quad \text{Indicates success.} \\
0 & \quad \text{Indicates failure.}
\end{align*}
\]

Keywords

\text{Add} — Add the specified multi-click handler for the ButtonPress event.

\text{Clicks} — The number of clicks required before the event handler is called. (Default: 2)

\text{Remove} — Remove the specified multi-click handler from the widget.

\text{TimeInterval} — The maximum time interval, in milliseconds, between multiple clicks that determines whether consecutive clicks are interpreted as multiple clicks.

Event Handler Parameters

The following parameters are required for the multi-click event handler procedure:

\text{wid} — The ID of the widget from which the multi-click event handler is called.
**userdata** — User-defined data.

**event** — An opaque event handle used in subsequent calls to request event data.

**Discussion**

WwMultiClickHandler allows an application to handle multiple mouse clicks (two, three, or more) when they are received in the given time interval.

WwMultiClickHandler is a special type of event handler that only handles mouse-button clicks. The mask ButtonPressMask is set internally and does not have to be set by the user as in the case of a generic event handler such as WwHandler.

**Example**

This example creates a drawing area and displays an image. The event handler DrawMClickHandler is activated when a mouse click occurs in the drawing area. The callback DrawCB displays an image in the drawing area.

Enter the event handler and callback procedures into a file, and compile it with the .RUN command. Then, enter the widget commands at the WAVE> prompt.

To dismiss the drawing area, select the appropriate function (such as Close) from the window manager menu.

**Event Handler and Callback Procedures**

```plaintext
PRO DrawMClickHandler, wid, shell, event
    ; This event handler takes action when a button is pressed while the pointer is
    ; in the drawing area.
    COMMON draw, top, img
    PRINT, WwGetPosition(event)
    ; Print the position of the button press.
    PRINT, WwGetButton(event, State = state)
    ; Print the button.
    PRINT, ‘State:’, state
    ; Print the state of the modifier keys.
    status = WwMultiClickHandler(wid, ‘DrawHandler’, /Remove)
    ; Remove the handler.
END

PRO DrawCB, wid, data
    ; This callback displays the image.
    COMMON draw, top, img
```
Widget Commands

```plaintext
COMMON draw, top, img

top=WwInit('ww_ex200', 'Examples', layout)
LOADCT, 5, /Silent
img=BYTARR(512,512)
OPENR,1, !Data_Dir + 'head.img'
READU,1,img
CLOSE, 1

draw=WwDrawing(layout, 1,'DrawCB', $ 
    [256,256], [512,512], Area = darea)

status = WwMultiClickHandler(darea, 'DrawMClickHandler', top)
    ; Register the multi-click (double-click in this case) handler.
status=WwSetValue(top, /Display)
WwLoop
```

See Also

WtAddHandler, WwHandler

---

**WwOptionMenu Function**

Creates an option menu.

**Usage**

```plaintext
option = WwOptionMenu(parent, label, items)
```

**Input Parameters**

- `parent` — The widget ID of the parent widget.
- `label` — A string containing the text of the option menu label. If this parameter is a null string, the *Name* keyword is used instead.
items — An unnamed structure specifying the option menu items. For more information, see *Creating and Handling Menus* on page 173.

**Returned Value**

option — The widget ID of the option menu box.

**Input Keywords**

Name — A two-element string array. The first element, name(0), contains the name of a pulldown widget; the second element, name(1), contains the name of the option menu.

The *Name* keyword can be used in place of the *label* parameter, although *label* (if other than a null string) will take precedence if both are given.

Position — If the option menu widget is to be placed in a bulletin board layout, use this keyword to specify the \( x, y \) coordinates of the option menu widget within the bulletin board.

**Output Keywords**

Menus — Returns an array of menu pane widget IDs in the order in which the menus were created. A menu pane is a special menu widget that serves as a container for a menu item. Menu pane widget IDs can be used in the *WwMenuItem* function to add, modify, or delete menu items.

**Color/Font Keywords**

Background — Specifies the background color name.

Font — Specifies the name of the font used for the button text.

Foreground — Specifies the foreground color name.

**Attachment Keywords**

Bottom — If a widget ID is specified (for example, `Bottom=wid`), then the bottom of the option menu widget is attached to the top of the specified widget. If no widget ID is specified (for example, `/Bottom`), then the bottom of the option menu widget is attached to the bottom of the parent widget.

Left — If a widget ID is specified (for example, `Left=wid`), then the left side of the option menu widget is attached to the right side of the specified widget. If no
widget ID is specified (for example, \texttt{\textbackslash Left})}, then the left side of the option menu widget is attached to the left side of the parent widget.

\textbf{Right} — If a widget ID is specified (for example, \texttt{Right=wid}), then the right side of the option menu widget is attached to the left side of the specified widget. If no widget ID is specified (for example, \texttt{\textbackslash Right}), then the right side of the option menu widget is attached to the right side of the parent widget.

\textbf{Top} — If a widget ID is specified (for example, \texttt{Top=wid}), then the top of the option menu widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, \texttt{\textbackslash Top}), then the top of the option menu widget is attached to the top of the parent widget.

\subsection*{Get/Set Value}

\textbf{getvalue} — The button label or icon pixmap ID for the selected menu button. Or, if the button is a toggle, returns whether it is selected or unselected (selected = 1, unselected = 0).

\textbf{setvalue} — The button label or icon pixmap ID for the selected menu button. Or, if the button is a toggle, selects or unselects the toggle button (select = 1, unselect = 0).

\subsection*{Callback Parameters}

Any option menu callback procedure must have the following two parameters:

\textbf{wid} — Widget ID of the selected menu item.

\textbf{index} — Index of the selected menu item (1 – \texttt{n}).

\subsection*{Discussion}

An option menu is a button that, when selected, displays a menu. The current selection is always displayed on the option menu button. When the user makes a selection, the option menu button is updated to reflect the change.

Pullright menus are not allowed in an option menu.

\subsection*{Example 1}

This example creates an option menu. The menu information is defined in an unnamed structure called \texttt{fonts}. The callback routine, \texttt{MenuCB}, is called whenever a menu item is selected.
Enter the callback procedure into a file, and compile the procedure with the .RUN command. Then, enter the widget commands at the WAVE> prompt. To dismiss the option menu, select the appropriate function (such as Close) from the window manager menu.

**Callback Procedure**

```idl
PRO MenuCB, wid, index
    PRINT, 'Menu Item', index, 'selected.'
    value = WwGetValue(wid)
    PRINT, value
END
```

**Widget Commands**

```idl
top = WwInit('ww_ex31', 'Examples', layout)
fonts = {,callback:'MenuCB', $
    button:'Adobe',$
    button:'Helvetica',$
    button:'Courier'}
opmenu = WwOptionMenu(layout, 'Fonts:', $
    fonts, Position=[0,150])
status = WwSetValue(top, /Display)
WwLoop
```

**Example 2**

A typical resource specification for a menu label used in WwOptionMenu is:

```
myapp.layout.option_menu.labelString: Style:
```

**See Also**

WwMenuItem, WwMenuBar, WwPopupMenu

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, *Using WAVE Widgets*.

For additional information on the color and font keywords, see *Setting Colors and Fonts* on page 201.

For additional information on attachment keywords, see *Form Layout: Attachments* on page 170.
WwPickFile Function

A convenience routine to easily create a modal file selection dialog that blocks until a file name has been selected. This routine will not return until a file name is selected or the Cancel button is pressed.

Usage

\[filename = \text{WwPickFile}(\text{parent} [, \text{HelpCallback} ] )\]

Input Parameters

\textit{parent} — The widget ID of the parent.

\textit{HelpCallback} — (Optional) A string containing the name of the callback routine that is called when the Help button is selected. The Help keyword can be used instead of the parameter.

Returned Value

\textit{filename} — A string containing the name of the file selected. If the Cancel button was selected a null string ("") is returned.

Keywords

\textit{Dir} — Specifies a string containing the directory path.

\textit{File} — Specifies a string containing the default file selection.

\textit{Help} — Use this keyword to specify a help topic when the HelpCallback parameter is not specified. This keyword can specify a two-element or one-element string array. If it is a two-element array, the first element is the name of a help topic and the second is the filename of the help file containing the topic. A one-element array specifies only the name of a help topic. In this case the default help file is used. For information on the location of the default help file, refer to the HELP command. The specified help topic is displayed in the online help viewer when the Help button is pressed.
**Name** — A string containing the name of the file selection box widget. The Name specified is the widget name used as part of the resource specification. (Default: file.)

**Pattern** — Specifies the search pattern used in combination with the directory in determining files to be displayed.

**Position** — Specifies the position of the upper-left corner of the file selection window on the screen in pixels.

**Shell_name** — Specifies the name of the TopLevelShell container widget used to hold the file selection box widget. The Shell_name specified is the top-level widget shell name used as part of the resource specification. (Default: fileshell.)

**Title** — Specifies a string containing the file selection widget’s title.

**Background** — Specifies the background color name.

**Font** — Specifies the name of the font used for text.

**Foreground** — Specifies the foreground color name.

### Discussion

The **Block** and **NonBlock** keywords found in the WwFileSelection widget are not available. Because of the method that is used to process widget events and return the filename, multiple instances of this dialog cannot exist. For this reason, the dialog always blocks.

### Example

```plaintext
PRO ButtonCB, wid, data
  file = WwPickFile(wid)
  PRINT, file
END

PRO PickTest
  top = WwInit('PickTest', 'PickTest', layout)
  button = WwButtonBox(layout, 'File Tool', 'ButtonCB')
  status = WwSetValue(top, /Display)
  WwLoop
END
```
See Also

WwFileSelection

**WwPopupMenu Function**

Creates a popup menu.

**Usage**

\[ \text{popup} = \text{WwPopupMenu}(\text{parent, items}) \]

**Input Parameters**

- **parent** — The widget ID of the parent widget.
- **items** — An unnamed structure specifying the menu bar items. For more details, see *Creating and Handling Menus* on page 173.

**Returned Value**

- **popup** — The widget ID of the popup menu widget.

**Keywords**

- **Menus** — Returns an array of menu pane widget IDs in the order in which the menus were created. A menu pane is a special menu widget that serves as a container for a menu item. Menu pane widget IDs can be used in the WwMenuItem function to add, modify, or delete menu items.
- **Name** — A string specifying the name of the PopupMenu widget. The Name specified is part of the resource specification. (Default: `menu`.)

**Color/Font Keywords**

- **Background** — Specifies the background color name.
- **Font** — Specifies the name of the font used for text.
- **Foreground** — Specifies the foreground color name.
Get/Set Value

getvalue — The button label or icon pixmap ID for the selected menu button. Or, if the button is a toggle, returns whether it is selected or unselected (selected = 1, unselected = 0).

setvalue — The button label or icon pixmap ID for the selected menu button. Or, if the button is a toggle, selects or unselects the toggle button (select = 1, unselect = 0).

Callback Parameters

Any popup menu callback procedure must have the following two parameters:

wid — Widget ID of the selected menu item.

index — Index of the selected menu item (1 – n).

Discussion

A popup menu is a menu that appears when the user presses a mouse button (usually the right button) when the pointer is in the popup menu’s parent widget. Thus, the popup menu is not tied to any menu button or menu bar.

NOTE Avoid creating popup menus that have a multi-line text widget as a parent. The system editing menu may appear over the popup menu that you have defined.

Example

This example creates a drawing widget that serves as the parent for a popup menu. The popup menu is activated when the right mouse button is pressed while the pointer is over the drawing widget. The callback procedure is executed when a menu button is selected.

Enter the callback procedures into a file, and compile the procedures with the .RUN command. Then, enter the widget commands at the WAVE> prompt (or enter them in a command file and run them with the @ command). To dismiss the button box, select the appropriate function (such as Close) from the window manager menu.

Callback Procedures

PRO DrawCB, wid, data
    COMMON draw, img
    PRINT, 'Draw'
Widget Commands

```wv
Tv, img

END

PRO MenuCB, wid, index
    PRINT, 'Menu Item', index, 'selected.'
    VALUE = WwGetValue(wid)
    PRINT, value
END

COMMON draw, img
LOADCT, 5, /SILENT
img = BYTARR(512,512)
OPENR, 1, !Data_DIR + 'head.img'
READU, 1, img
CLOSE, 1
draw = WwDrawing(layout, 1,'DrawCB', $
    [256,256], [512,512])
status = WwSetValue(top, /Display)
menus = {,callback:'MenuCB', $
    menubutton:'Fonts', $
    menu:{,callback:'MenuCB', $
        menubutton:'Adobe', $
        menu:{,callback:'MenuCB', $
            toggle:'Normal', $
            toggle:'Bold', $
            toggle:'Italic', $
            button:'Helvetica', $
            button:'Courier'}, $
        button:'Adobe', $
        menu:{,callback:'MenuCB', $
            toggle:'Normal', $
            toggle:'Bold', $
            toggle:'Italic', $
            button:'Helvetica', $
            button:'Courier'}, $
        button:'Adobe', $
        menu:{,callback:'MenuCB', $
            toggle:'Normal', $
            toggle:'Bold', $
            toggle:'Italic', $
            button:'Helvetica', $
            button:'Courier'}}}
menu = WwPopupMenu(draw, menus)
status = WwSetValue(top, /Display)
WwLoop
```
See Also

WwMenuItem, WwMenuBar, WwOptionsMenu

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, *Using WAVE Widgets*.

For additional information on the color and font keywords, see *Setting Colors and Fonts* on page 201.

For information on Get and Set values, see *Setting and Getting Widget Values* on page 206.

---

**WwPreview Procedure**

Creates an ASCII data preview widget.

**Usage**

WwPreview, *parent, confirmCallback, clearCallback*

**Input Parameters**

*parent* — The widget ID of the parent widget.

*confirmCallback* — A procedure called when an area selection is confirmed.

*clearCallback* — A procedure called when an area selection is cleared.

**Keywords**

*AutoDefine* — If nonzero, the header, record, and field areas are automatically defined after the file is loaded.

*Filename* — A string containing the name of the file from which column-oriented ASCII data is read.

*Format* — A string specifying the format used to process the data when the *Auto-Define* keyword is set. Format choices are: ‘FIXED’ or ‘FREE’. (Default: ‘FIXED’)

*Name* — A string containing the name of the preview widget. (Default: *preview*)

*Nlines* — Specifies the number of lines to read from the data file.
**Position** — If set, specifies a two-element vector containing the x, y coordinates for the widget within a bulletin board layout.

**Selection** — A string specifying the type of area to select initially in the preview window. The cursor changes to an H, R, or F to reflect the type of area to be selected: Header, Record, or Field. The choices for this keyword are: ‘ANY’, ‘HEADER’, ‘RECORD’, or ‘FIELD’.

**Visible** — A two-element vector specifying the number of rows and columns displayed. If the data file size is bigger than the number of visible rows and columns in the preview window, scrollbars are placed at the right and bottom edges of the window. (Default: [5, 5])

**Color/Font Keywords**

**Background** — Specifies the background color name.

**Font** — Specifies the name of the font used for text. Only nonproportional (fixed-width) fonts are supported.

**Foreground** — Specifies the foreground color name.

**Attachment Keywords**

**Bottom** — If a widget ID is specified (for example, Bottom=wid), then the bottom of the widget is attached to the top of the specified widget. If no widget ID is specified (for example, /Bottom), then the bottom of the widget is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, Left=wid), then the left side of the widget is attached to the right side of the specified widget. If no widget ID is specified (for example, /Left), then the left side of the widget is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, Right=wid), then the right side of the widget is attached to the left side of the specified widget. If no widget ID is specified (for example, /Right), then the right side of the widget is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, Top=wid), then the top of the widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, /Top), then the top of the widget is attached to the top of the parent widget.
Get/Set Value

getvalue — Gets the contents of the ASCII data file.

setvalue — Sets the contents of the ASCII data file.

Confirm Callback Parameters

An area selection is committed after it is confirmed by the user. Usually, confirmation occurs when the user clicks MB3. When the user confirms an area selection, a callback routine is called. This callback must have the following parameters:

- **wid** — The preview widget ID.
- **object** — A string containing the type of area to be committed. Valid areas are: ‘HEADER’, ‘RECORD’, ‘FIELD’.
- **status** — Indicates whether WtPreview has already flagged an error. If status is nonzero, an error detection has occurred. For example, if the area selected by the user is invalid, status is nonzero.
- **area** — A four-element vector specifying the exact lines and columns of the preview window to commit. This vector specifies the following positions: the starting line, starting column, ending line, and ending column.
- **doit** — An output parameter that gives the go-ahead to commit the selection. A value of 0 indicates that the selection should be refused; a nonzero value means the selection is accepted.

Clear Callback Parameters

A routine that is called when an area selection is cleared must have the following parameters:

- **wid** — The preview widget ID.
- **object** — The type of area— ‘HEADER’, ‘RECORD’, or ‘FIELD’— to be cleared.
- **area** — A four-element vector specifying the exact lines and columns of the preview window to clear. This vector specifies the following positions: the starting line, starting column, ending line, and ending column.
Discussion

WwPreview lets you create WAVE Widget tools that allow the user to look at the structure of a column-oriented ASCII data file and interactively select specific regions of the file. The three primary parts of a file that can be selected are the header, records, and fields.

- **Header** — One or more lines at the beginning of the data file that are ignored because they do not contain meaningful data.
- **Record** — The smallest portion of the file that contains a set of data (one value from each field). A record can be defined to be exactly one line, less than one line, or more than one line of the data file. A record must start in the first column of the data file and must be on the line immediately following the header.
- **Field** — A specific area in the file that contains a value and is contained in the record.

The preview widget also allows these regions to be defined automatically when the file is opened, via the *AutoDefine* keyword.

WwPreview provides the following mechanisms to select headers, records, and fields:

- **StartSelection** — Marks the beginning of the specific area. By default, a selection is started when the user clicks MB1.
- **EndSelection** — Marks the end of the specific area. By default, a selection is ended when the user clicks MB2.
- **ConfirmSelection** — Confirms the current selection. By default, a selection is confirmed when the user clicks MB3.

Example

This example application uses the preview widget to create a tool to read a data file and allow the user to select the header, record, and field regions of the data file. The data file can be any column-oriented ASCII file. This example program automatically loads the following data file into the preview window:

(UNIX)  
<wavedir>/data/air_qual.dat

(OpenVMS)  <wavedir>: [DATA] AIR_QUAL.DAT

(Windows) <wavedir>\data\air_qual.dat

Where <wavedir> is the main PV-WAVE directory.
**NOTE** The application in this example uses the WAVE Widgets function WwPreviewUtils to provide some additional functionality not available with WwPreview, such as clearing previously defined areas and setting characters used for filtering and data separation.

The following illustration shows the preview window created in this example:

![Preview Window Illustration](image)

**Figure 7-1** The preview window created in this example displays a data file and allows the user to select header, record, and field regions using the mouse.

```
PRO ConfirmSelected, wid, object, status, area, doit
    PRINT, 'object confirmed:', object, 'status:', status
    PRINT, 'area:', area
    PRINT, 'doit:', doit
END

PRO ClearSelected, wid, object, area
    PRINT, 'object cleared:', object
    PRINT, 'area:', area
END
```
PRO FileOK, wid, fsshell
  COMMON PREVIEW_Widgets, top, shell, preview
  filename = WwGetValue(wid)
  OPENR, /GET_LUN, unit, filename
  source = ''
  line = ''
  WHILE NOT eof(unit) DO BEGIN
    READF, unit, line
    source = source + line + '\012'
  ENDWHILE
  FREE_LUN, unit
  status = WwSetValue(preview, source)
  status = WwSetValue(fsshell, /Close)
END
PRO FileDone, wid, fsshell
  status = WwSetValue(fsshell, /Close)
END
PRO FileMenuCB, wid, index
  @wtxmconsts
  COMMON PREVIEW_Widgets, top, shell, preview
  CASE index OF
    1: BEGIN
      file = WwFileSelection(wid, 'FileOK', 'FileDone', $
        TITLE = 'Search', $
        DIR = !Dir+'/data/')
      END
    2: BEGIN
      status = WwSetValue(top, /Close)
    END
  ENDCASE
END
PRO EditMenuCB, wid, index
  @wtxmconsts
  COMMON PREVIEW_Widgets, top, shell, preview
  CASE index OF
    1: BEGIN
      status = WwPreviewUtils(preview, /Clear, $
        /All)
    END
    2: BEGIN
      status = WwPreviewUtils(preview, 'RECORD', $
1, [0, 54, 3, 3], /Clear) 
END 
3: BEGIN 
header = [0, 53, 0, 2] 
status = WwPreviewUtils(preview, 'HEADER', 1, $ 
header, /Commit) 
record = [0, 54, 3, 3] 
status = WwPreviewUtils(preview, 'RECORD', $ 
record, /Commit) 
END 
ENDCASE 

END 

PRO SeparatorsOK, wid, index 
COMMON PREVIEW_Widgets, top, shell, preview 
status = WtSet(preview, {, $ 
separatorChars:WwGetValue(wid) 
}) 
END 

PRO FiltersOK, wid, index 
COMMON PREVIEW_Widgets, top, shell, preview 
status = WtSet(preview, {, filterChars:WwGetValue(wid) 
}) 
END 

PRO OptionsMenuCB, wid, index 
COMMON PREVIEW_Widgets, top, shell, preview 
CASE index OF 
  2: BEGIN 
  separ = WtGet(preview, 'separatorChars') 
  status = WwDialog(preview, 'Separators:', $ 
  'SeparatorsOK', Text = separ) 
END 

  3: BEGIN 
  filter = WtGet(preview, 'filterChars') 
  status = WwDialog(preview, 'Filters:', $ 
  'FiltersOK', Text = filter) 
END 
ENDCASE 

END 

PRO FormatMenuCB, wid, index 
@wtxmconsts 
COMMON PREVIEW_Widgets, top, shell, preview 
CASE index OF 
  1: BEGIN 
  status = WtSet(preview, {, $ 
  formatType:XmFORMAT_FIXED})
END
2: BEGIN
status = WtSet(preview, {, $
formatType:XmFORMAT_FREE})
END
ENDCASE
END

PRO ButtonCB, wid, data
@wtxmconsts
COMMON PREVIEW_Widgets, top, shell, preview
CASE data OF
1: BEGIN
status = WwPreviewUtils(preview, /AutoDefine)
END
2: BEGIN
status = WtSet(preview, {, $
selectionMode:XmHEADER_PW_OBJECT})
END
3: BEGIN
status = WtSet(preview, {, $
selectionMode:XmRECORD_PW_OBJECT})
END
4: BEGIN
status = WtSet(preview, {, $
selectionMode:XmFIELD_PW_OBJECT})
END
5: BEGIN
status = WtSet(preview, {, $
selectionMode:XmANY_PW_OBJECT})
END
ENDCASE
END

PRO test_preview
@wtxmclasses
@wtxmconsts
COMMON PREVIEW_Widgets, top, shell, preview
top = WwInit('preview', 'Preview', layout, $
Title = 'Preview', Background = 'White', $
Foreground = 'red', $
/FORM, BORDER = 3, Position = [100, 100])
menus = {, menubutton:'File', $
menu:{, callback:'FileMenuCB', $
button:'Open', $
button:'Quit'}, $
menubutton:'Edit', $
menu:{, callback:'EditMenuCB', $
button:'Clear All', $
button:'Clear', $
button:'Set areas'}, $
menubutton:'Options', $
menu:{, callback:'OptionsMenuCB', $
menubutton:'Format', $
menu:{, callback:'FormatMenuCB', $
button:'Fixed', $
button:'Free'}, $
button:'Separators...', $
button:'Filter...'} $
}
bar = WwMenuBar(layout, menus, /Left, $
/Right, /Top, $
Font = '-adobe-times-bold-r-normal--14*')

preview = WwPreview(layout, ‘ConfirmSelected’, $
’ClearSelected’, $
/Auto, File = ‘air_qual.dat’, $
Top = bar, /Left, /Right, $
Visible = [20, 60], Font = font, $
Foreground = ‘Blue’, $
Background = ‘LightGrey’) 
labels = [‘Auto Define’, ‘Header’, ‘Record’, $
‘Fields’, ‘Any’]
bbox = WwButtonBox(layout, labels, ‘ButtonCB’, $
/Horizontal, SPACING = 5, $
Top = preview, /Left, /Right, /Bottom)
status = WwSetValue(tcp, /Display)

WwLoop

END

See Also

WtPreview, WwPreviewUtils
**WwPreviewUtils Function**

Manages the contents of a preview widget.

**Usage**

\[status = \text{WwPreviewUtils}(wid[,\ param1,\ param2,\ param3])\]

**Input Parameters**

- **wid** — The widget ID of the WwPreview widget.
- **param1** — (optional) This parameter depends on keyword use in the function calling sequence. See **Keywords** for more information.
- **param2** — (optional) This parameter depends on keyword use in the function calling sequence. See **Keywords** for more information.
- **param3** — (optional) This parameter depends on keyword use in the function calling sequence. See **Keywords** for more information.

**Returned Value**

- **status** — A value indicating success or failure of the function.

  - 1 Indicates success.
  - 0 Indicates failure.

**Keywords**

- **AutoDefine** — If nonzero, automatically defines the header, record, and field regions of the ASCII data file.
- **Clear** — If nonzero, clears all currently selected regions in the preview widget. If the **All** modifier keyword is also specified, all regions of the ASCII data file are cleared. If **All** is not specified, you must specify the following parameters:

  - **param1** — An array of strings containing the types of areas to clear. These types include: [‘Header’, ‘Record’, ‘Field’].
  - **param2** — The number of areas to clear.
  - **param3** — The areas to clear.
Commit — If nonzero, commits all currently selected regions in the preview widget. If you use this keyword, you must specify the following parameters:

- **param1** — An array of strings containing the types of areas to commit. These types include: ['Header', 'Record', 'Field'].
- **param2** — The number of areas to commit.
- **param3** — The areas to commit.

Format — If specified, selects the datatype format from the following formats; 'FIXED', or 'FREE'. (Default: 'FIXED')

Select — If nonzero, selects the specified regions. If this keyword is used, you must specify the following parameters:

- **param1** — An array of strings containing the types of areas to select. These types include: ['Header', 'Record', 'Field'].
- **param2** — The areas to be selected.

SelectionMode — If specified, sets the mode for the selection area to one of the following: 'HEADER', 'RECORD', 'FIELD', 'ANY'.

SeparatorChars — A string containing the characters to use as field separators. For example, to use comma and semicolon as separators, the string would be ',;'.

Value — Sets the contents of the preview window to the specified string.

See Also

WwPreview

---

**WwRadioBox Function**

Creates a box containing radio buttons.

Usage

```
radio = WwRadioBox(parent, [labels,] callback)
```

Input Parameters

- **parent** — The widget ID of the parent widget.
labels — (optional) A string array of button labels. If the labels parameter is not defined or is an array of null strings, the function looks for the button labels in a resource specification (see Discussion).

callback — A string containing the name of the callback that is executed when a radio button is selected or unselected.

**NOTE** The callback procedure is called twice if the Oneofmany keyword is specified. It is called once when the previously select button is unselected and again for the newly selected button. Use WwGetValue to determine if a button is currently selected or unselected.

**Returned Value**

radio — The widget ID of the radio box widget when more than one radio button is used. If only one button is requested, the widget ID of the layout around that one button is returned. To get the widget ID of that single button, use the Toggles output keyword.

**Keywords**

AlignLeft — If nonzero, aligns the text to the right of the toggle to be left-justified. This keyword is only useful if you have multiple toggle buttons that are vertically oriented. By default, buttons are center-justified.

AlignRight — If nonzero, aligns the text to the right of the toggle to be right-justified. This keyword is only useful if you have multiple toggle buttons that are vertically oriented. By default, buttons are center-justified.

Border — Specifies the width in pixels of the radio box and button borders.

Center — An array specifying the position of the left and right edge of buttons as a percentage of radio box width. By default, buttons are spaced evenly in the box.

Form — When present and nonzero, buttons are placed in a form layout and all specified attachment keywords are honored (i.e., /Left, /Right, /Top, /Bottom). By default, buttons are placed in a row/column layout.

Horizontal — When present and nonzero, creates a horizontally aligned row of radio buttons.

Layout_name — Specifies the name of the layout widget. This name is part of the resource specification. Layout_name names the Form widget (if Form is specified),
the SimpleRadioBox widget (if Oneofmany is specified), or the SimpleCheckBox widget (if Noofmany is specified). (Default: radio.)

**Measure** — Specifies the number of columns of radio buttons (for a vertical box) or rows (for a horizontal box).

**Name** — A string array specifying the button widget names. The button widget names specified are part of the resource specifications containing the button labels. The Name keyword can be used in place of the labels parameter, although labels (if other than an array of null strings) will take precedence if both are given. (Default: button_1, button_2, ..., button_n.)

**Noofmany** — When present and nonzero, creates nonexclusive radio buttons, where any number of buttons can be selected at once.

**Oneofmany** — When present and nonzero, creates exclusive radio buttons, where only one button can be selected at a time.

**Position** — If the radio box widget is to be placed in a bulletin board layout, use this keyword to specify the x, y coordinates of the radio box widget within the bulletin board.

**Spacing** — Specifies the space in pixels between radio buttons.

**Toggles** — Returns an array of toggle button widget IDs.

**Vertical** — When present and nonzero, creates a vertically aligned column of radio buttons.

**Color/Font Keywords**

**Background** — Specifies the background color name.

**Basecolor** — Specifies the base color.

**Font** — Specifies the name of the font used for text.

**Foreground** — Specifies the foreground color name.

**Attachment Keywords**

**Bottom** — If a widget ID is specified (for example, Bottom=wid), then the bottom of the radio box widget is attached to the top of the specified widget. If no widget ID is specified (for example, /Bottom), then the bottom of the radio box widget is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, Left=wid), then the left side of the radio box widget is attached to the right side of the specified widget. If no wid-
get ID is specified (for example, /Left), then the left side of the radio box widget is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, Right=wid), then the right side of the radio box widget is attached to the left side of the specified widget. If no widget ID is specified (for example, /Right), then the right side of the radio box widget is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, Top=wid), then the top of the radio box widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, /Top), then the top of the radio box widget is attached to the top of the parent widget.

**Get/Set Value**

*getvalue* — Gets the state of the selected radio button. Set = 1; unset = 0.

*setvalue* — Sets the state of the selected radio button. Set = 1; unset = 0.

**Callback Parameters**

Any radio button callback procedure must have the following two parameters:

*wid* — Toggle button widget ID.

*index* — Index of the toggle button changed (1 – n).

**Discussion**

This function creates a box containing a specified number of rows or columns of labeled toggle buttons. If only one button is created, a box is not created; instead, only a single button widget is created.

Part of the button label resource specification can be specified using the *Name* keyword, otherwise the default is the *button_n.labelString* resource (where labelString is the attribute).

**TIP** The *labels* parameter provides a method for “hard-coding” the button labels in the application. For greater flexibility, create a resource file containing the button labels using a text editor, and load the button label resources using WtResource. The *Name* keyword can then be used in the WwRadioBox calling sequence to specify the resources.
Example 1
This example creates a box containing radio buttons. The callback is executed whenever the user clicks on a button.

Enter the callback procedure into a file, and compile the procedure with the .RUN command. Then, enter the widget commands at the WAVE> prompt. To dismiss the radio button box, select the appropriate function (such as Close) from the window manager menu.

Callback Procedure
PRO RadioCB, wid, which
   CASE which OF
      1: PRINT,’First Toggle Selected’
      2: PRINT,’Second Toggle Selected’
      3: PRINT,’Third Toggle Selected’
   ENDCASE
   value = WwGetValue(wid)
   PRINT, value
END

Widget Commands
top = WwInit(‘ww_ex33’, ‘Examples’, layout)
labels = [‘System’, ‘Owner’, ‘Group’]
rbox = WwRadioBox(layout, labels, ’RadioCB’, $ /Vertical, Border=2, Spacing=20)
status = WwSetValue(top, /Display)
WwLoop

Example 2
A typical resource specification for a button label used in WwRadioBox is:
myapp*radio.shade_g.labelString: Gouraud

See Also
For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, Using WAVE Widgets.
For additional information on the color and font keywords, see Setting Colors and Fonts on page 201.
For additional information on attachment keywords, see Form Layout: Attachments on page 170.

For information on Get and Set values, see Setting and Getting Widget Values on page 206.

---

**WwResource Function**

Queries, creates, saves, or modifies the widget resource database.

**Usage**

\[ \text{value} = \text{WwResource(} \text{resvar} \text{)} \]

**Input Parameters**

`resvar` — (optional) A string containing a resource variable specification in the resource database.

**Returned Value**

`value` — The returned value depends on the input parameter and/or the use of keywords as shown in the following table.

<table>
<thead>
<tr>
<th>Value Returned</th>
<th>Parameter or Keyword Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>A string containing the value associated with <code>resvar</code>; a null string, if <code>resvar</code> isn't found; or the default value specified.</td>
<td><code>resvar</code>, or <code>resvar</code> with <code>Default</code> keyword</td>
</tr>
<tr>
<td>An integer value of 1 indicating success, or 0 indicating failure.</td>
<td><code>Add</code>, <code>Load</code>, or <code>Save</code> keywords</td>
</tr>
<tr>
<td>A string containing the resource specification a widget.</td>
<td><code>Spec</code> keyword</td>
</tr>
</tbody>
</table>

**Keywords**

`Add` — A string containing a name: value resource specification to merge into the resource database in the current session. If the resource specification already exists in the application, the `Add` keyword takes precedence.
**Default** — (Used only if resvar is specified.) A string specifying a default value for the resource variable in resvar. If resvar doesn’t match anything in the resource database, the default value is returned.

**Load** — A string specifying the pathname of a resource file to be merged with the existing resource database. If the resource file was specified in the application, the use of the *Load* keyword takes precedence.

**Save** — A string specifying the pathname of the resource file in which to save the currently defined resources in the resource database. If the specified file already exists, the contents are overwritten.

**Spec** — Used to specify a widget ID. The resource variable specification of the widget is returned.

**Discussion**

WwResource provides direct access to the widget resource database from WAVE Widgets applications. Resources which are loaded or added using WwResource are then automatically used by all subsequently created WAVE Widgets. The resource specifications merged into the resource database using either the *Load* or *Add* keywords supercede existing definitions.

**Examples**

The following example shows how a resource specification added during a session takes precedence over the existing definition.

```plaintext
myapp*background: red

; The existing resource specification in the database.

value = WwResource(ADD='myapp*background: blue')

; The *Add* keyword changes the color to blue instead. For the rest of the
; session, or until it is redefined, the background color will be blue.
```

In the next example, the syntax for using the *Load* keyword is illustrated.

```plaintext
value = WwResource(LOAD='/usr/mydir/myapp/myapp.ad')

; This merges the resources found in the specified file with the
; existing resource database.
```

This example shows how to request a value associated with a resource name.

```plaintext
value = WwResource('myapp*mybutton.label', $
    Default='my label')

; This usage returns the value associated with the resource variable
; specified. The *Default* keyword is used to return the value ‘my label’,
; if the named resource doesn’t exist.
```
An example for writing all defined resources to a file is:

```python
value = WwResource(SAVE='/usr/mydir/myapp/myapp.newad')
```

To request the resource variable specification of a particular widget identified by its associated widget ID, use:

```python
value = WwResource(SPEC=widget_id)
```

**See Also**

- WtResource
- WoLoadResources

---

### WwSeparator Function

Creates a horizontal or vertical line that separates components in a graphical user interface.

**Usage**

```python
separator = WwSeparator(parent)
```

**Input Parameters**

- `parent` — The parent widget’s ID.

**Returned Value**

- `separator` — The separator widget’s ID.

**Input Keywords**

- **Double_Dashed_Line** — If present and nonzero, creates a double dashed-line separator.
- **Double_Line** — If present and nonzero, creates a double-line separator.
- **Height** — Specifies the height (in pixels) of vertical separators.
- **Horizontal** — If present and nonzero, creates a horizontal separator. (Default)
- **Margin** — An integer specifying the spacing on each end of the separator, in pixels. For a horizontal separator, the margin is added on the left and right. For a vertical separator, the margin is added on the top and bottom.
**Name** — A string specifying the name of the separator widget. This name is part of the resource specification for the separator. (Default: `separator`)

**No_LINE** — If present and nonzero, creates a separator with no line.

**Position** — If the separator is to be placed in a bulletin board layout, use this keyword to specify the x, y coordinates of the separator within the bulletin board.

**Single_Dashed_Line** — If present and nonzero, creates a single dashed-line separator.

**Single_Line** — If present and nonzero, creates a solid single-line separator.

**Shadow_Etched_In** — If present and nonzero, creates a separator that appears to be etched inward. (Default)

**Shadow_Etched_Out** — If present and nonzero, creates a separator with a raised appearance.

**Vertical** — If present and nonzero, creates a vertical separator. (Default: horizontal)

**Width** — Specifies the width, in pixels, of a horizontal separator.

**Color/Font Keywords**

**Background** — Specifies the background color name.

**Foreground** — Specifies the foreground color name.

**Attachment Keywords**

**Bottom** — If a widget ID is specified (for example, `Bottom=wid`), then the bottom of the separator widget is attached to the top of the specified widget. If no widget ID is specified (for example, `/Bottom`), then the bottom of the separator widget is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, `Left=wid`), then the left side of the separator widget is attached to the right side of the specified widget. If no widget ID is specified (for example, `/Left`), then the left side of the separator widget is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, `Right=wid`), then the right side of the separator widget is attached to the left side of the specified widget. If no widget ID is specified (for example, `/Right`), then the right side of the separator widget is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, `Top=wid`), then the top of the separator widget is attached to the bottom of the specified widget. If no widget ID is...
specified (for example, /Top), then the top of the separator widget is attached to the top of the parent widget.

**Get/Set Value**

None.

**Callback Parameters**

None.

**Discussion**

The separator widget resource name can be specified with the Name keyword, otherwise the default resource name is separator.

Use the Width and Height keywords when placing separators in a bulletin board layout (or, on Microsoft Windows platforms, a row/column layout). The Width keyword has no effect on vertical separators, and Height has no effect on horizontal separators. Microsoft Windows does not support the dashed line separators.

**Example 1**

This example generates all of the different types of separators that can be created with WwSeparator.

```plaintext
top = WwInit('sep_test', 'WidgetTest', layout, /Vertical, Spacing=15)

l = WwText(layout, Text='Sample Separators', /Label)

l = WwSeparator(layout, Width=100) ; Default separator--
    ; same as /Shadow_Etched_In
l = WwSeparator(layout, /Single_Line, Width=100)
l = WwSeparator(layout, /Single_Dashed_Line, Width=100)
l = WwSeparator(layout, /Double_Line, Width=100)
l = WwSeparator(layout, /Double_Dashed_Line, Width=100)
l = WwSeparator(layout, /Shadow_Etched_Out, Width=100)

status = WwSetValue(top, /Display)
```

**WwSetCursor Function**

Sets the cursor for a widget.

**Usage**

\[ status = WwSetCursor(wid, cursor) \]

**Input Parameters**

- `wid` — The widget ID.
- `cursor` — The name of the cursor or the cursor index. For a list of cursors, see Appendix C, *Widget Toolbox Cursors*.

**Returned Value**

- `status` — A value indicating success or failure of the function.
  - 1 Indicates success.
  - 0 Indicates failure.

---

**See Also**

For detailed information on GUI development, refer to the PV-WAVE Application Developer’s Guide.

**Figure 7-2** The six types of separators that can be created with WwSeparator.
Keywords

Default — If nonzero, sets the default cursor for the widget.

Wait — If nonzero, sets the wait cursor for the widget.

Example

For an example of WwSetCursor, see the example for the WwGetPosition function.

See Also

WtCursor

WwSetValue Function

Sets the specified value for a given widget.

Usage

\[
status = \text{WwSetValue}(\text{widget}, \{\text{value}\})
\]

Input Parameters

\text{widget} — The ID of the widget whose value you want to set.

\text{value} — The value of the widget (optional when keywords are specified).

Returned Value

\text{status} — Returns 1 if the function is successful, or 0 if the function is not successful.

Keywords

Close — If present and nonzero, closes the widget hierarchy from the top-level shell down.

Display — If present and nonzero, displays the widget hierarchy from the top-level shell down.

Hide — If present and nonzero, hides the specified widget or group of widgets (layout).
**Nonsensitive** — If present and nonzero, sets the widget to nonsensitive.

**Nopropagate** — If nonzero, suppresses the propagation of the value of **Userdata** for children of the container widgets (Form, Board, Row/Column).

**Position** — If the widget is to be placed in a bulletin board layout, use this keyword to specify a 2-element array containing the x,y coordinates of the widget within the bulletin board (in pixels).

**Scroll** — Specifies a 2-element array containing the scroll position (x,y coordinates, in pixels) of a drawing area widget or a text widget. The widget must be displayed before the call to WwSetValue with the **Scroll** keyword is made.

**Show** — If present and nonzero, shows the specified widget or group of widgets (layout).

**Sensitive** — If present and nonzero, sets the widget to sensitive.

**Size** — A two-element array specifying the width and height of the specified widget.

**Update** — If present and nonzero, all pending exposure (i.e., window repair) events are processed immediately. If you suspect that a callback procedure will take a long time, use this keyword to update the display before starting the time consuming operation.

**Userdata** — Stores the specified variable with the widget. By default, the value of **Userdata** is also set (propagated) for all children of the container widgets (Form, Board, Row/Column). To suppress the propagation of the value of **Userdata** use the **Nopropagate** keyword. You can retrieve the value of **Userdata** later with WwGetValue.

**Color/Font Keywords**

**Background** — Specifies the background color name.

**Font** — Specifies the name of the font used for text.

**Foreground** — Specifies the foreground color name.

**Attachment Keywords**

**Bottom** — If a widget ID is specified (for example, **Bottom=wid**), then the bottom of the widget is attached to the top of the specified widget. If no widget ID is specified (for example, **/Bottom**), then the bottom of the widget is attached to the bottom of the parent widget.
**Left** — If a widget ID is specified (for example, `Left=wid`), then the left side of the widget is attached to the right side of the specified widget. If no widget ID is specified (for example, `/Left`), then the left side of the widget is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, `Right=wid`), then the right side of the widget is attached to the left side of the specified widget. If no widget ID is specified (for example, `/Right`), then the right side of the widget is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, `Top=wid`), then the top of the widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, `/Top`), then the top of the widget is attached to the top of the parent widget.

**Discussion**

See the *Get/Set Value* section under each WAVE Widget function description to find out what value is set by `WwSetValue` for each function. For example, `WwSetValue` called with the ID of a list widget sets the current list of items to a specified string array of new items.

**Example**

The following example demonstrates two common uses of `WwSetValue`: displaying and closing widgets. In the callback routine `CommandDone`, `WwSetValue` is called to close the widget when the user selects the `Close` function from the window manager menu. In the *Widget Commands* section, `WwSetValue` is used to display the button box widget.

```plaintext
PRO CbuttonCB, wid, data
  command = WwCommand(wid, 'CommandOK', $
  'CommandDone', Position=[300,300], $
  Title='Command Entry Window')
END

PRO CommandOK, wid, shell
  value = WwGetValue(wid)
  PRINT, value
END

PRO CommandDone, wid, shell
  status = WwSetValue(shell, /Close)
END
```
**Widget Commands**

```plaintext
top = WwInit('ww_ex34', 'Examples', layout)
button = WwButtonBox(layout, 'Command', 'CbuttonCB')
status = WwSetValue(top, /Display)
WwLoop
```

**See Also**

For more information about how to write an application program based on WAVE Widgets, refer to *Chapter 5, Using WAVE Widgets*.

For additional information on the color and font keywords, see *Setting Colors and Fonts on page 201*.

For additional information on attachment keywords, see *Form Layout: Attachments on page 170*.

---

**WwTable Function**

Creates an editable 2D array of cells containing string data, similar to a spreadsheet.

**Usage**

```plaintext
table = WwTable(parent, callback [, variable])
```

**Input Parameters**

- **parent** — The widget ID of the parent widget.
- **callback** — A procedure that is called when the contents of a cell are modified.

**NOTE** It is the calling routine’s responsibility to modify the appropriate variables when a table cell has been modified.

- **variable** — (optional) A variable used to initialize the table cells. When this parameter is not supplied, the cells are initially empty, and the size of the table is set to the size specified by the `Cols` and `Rows` keywords. You can use the following types of variables with WwTable:
• scalar
• vector
• 2D array
• structure with scalar fields or date/time structure field

All values are converted to type string within WwTable.

**Returned Value**

*table* — The widget ID of the table widget.

**Keywords**

*Alignments* — A one-dimensional array (0, ..., *cols*–1) of column alignments. Valid values are:

0  Align cell contents to cell’s left edge (left justify).
1  Center cell contents (center justify).
2  Align cell contents to cell’s right edge (right justify).

*Labels* — A one-dimensional string array (0, ..., *cols*–1) of column labels.

**NOTE** It is recommended that you specify row/column labels. Clicking MB2 on a row label selects the whole row; clicking MB2 on a column label selects the whole column.

*Colors* — A two-dimensional array (0, ..., *rows*–1, 0, ..., *cols*–1) of color indexes for table cells. This enables you to highlight particular groups of cells.

*Cols* — The number of columns in the table. If the *Cols* keyword is not specified, and the *variable* parameter is specified, the number of columns is calculated from the dimensions of *variable*. If neither *Cols* nor *variable* are specified, the size of the table is set to one column.

*Cwidth* — A one-dimensional array (0, ..., *cols*–1) of column widths. If not specified, the default column width is 10 characters.

*Fixrows* — The number of fixed rows in the table. Fixed rows are non-scrollable, non-editable cells that can serve as labels.
**Fixcols** — The number of fixed columns in the table. Fixed columns are non-scrol-
lable, non-editable cells that can serve as labels.

**NOTE**  You can specify *Fixrows* or *Fixcols* for a table, but not both.

**Horizontal** — If this keyword is present and nonzero, the contents of *variable* are
displayed in natural fashion, i.e., *variable* rows are horizontal and columns are ver-
tical. *Horizontal* is enabled by default. The *Horizontal* and *Vertical* keywords are
mutually exclusive.

**Name** — A string specifying the name of the XbaeMatrix widget. The *Name* spec-
ified identifies the table widget name as part of the resource specification. *Name*
can be used to define the column and row attributes in place of the *Clabels* and *Rla-
bels* keywords, although those keywords will take precedence if specified in
addition to *Name*. (Default: *table*)

**Position** — If the widget is to be placed in a bulletin board layout, use this keyword
(a two-element vector) to specify the *x, y* coordinates of the widget within the bul-
letin board.

**Rlabels** — A one-dimensional string array (0, ..., *rows*–1) of row labels.

**Rows** — The number of rows in the table. If the *Rows* keyword is not specified, and
the *variable* parameter is specified, the number of rows is calculated from the
dimensions of *variable*. If neither *Rows* nor *variable* are specified, the size of the
table is set to one row.

**Setcelldata** — A copy of any PV-WAVE variable to be passed as client data to the
*Setcells* function.

**Setcells** — Allows you to name a PV-WAVE function that is responsible for setting
the values of the exposed cells. This function is called, like a callback, whenever
cells are exposed (for example, as the user scrolls through the table). The function
returns a string that is used to set the cell’s new value. The function’s input param-
ters are the table widget ID, a vector [row, column] specifying the exposed cell,
and client data specified with the *Setcelldata* keyword. If this function is specified,
the *variable* parameter is ignored.

Using *Setcells* can improve performance when a large table is created. Ordinarily,
when the table’s cells are populated by values taken from a single variable (the
*variable* parameter), WwTable actually copies this variable first. When *Setcells* is
used, only the values currently displayed are held in memory. See *Example 2* on
page 345 for more information on *Setcells*. 
**Vertical** — If this keyword is present and nonzero, the contents of *variable* are displayed as transposed (*variable* rows are vertical and columns are horizontal). The *Horizontal* and *Vertical* keywords are mutually exclusive.

**Visible** — A two-element vector specifying the number of rows and columns displayed. If the table size is bigger than the number of visible rows and columns, scrollbars are placed at the right and bottom edges of the window, so that you can view different portions of the table. If this keyword is not specified, four rows and four columns are displayed.

**Color/Font Keywords**

**Background** — Specifies the background color name.

**Font** — Specifies the name of the font used for text.

**Foreground** — Specifies the foreground color name.

**Attachment Keywords**

**Bottom** — If a widget ID is specified (for example, Bottom=wid), then the bottom of the widget is attached to the top of the specified widget. If no widget ID is specified (for example, /Bottom), then the bottom of the widget is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, Left=wid), then the left side of the widget is attached to the right side of the specified widget. If no widget ID is specified (for example, /Left), then the left side of the widget is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, Right=wid), then the right side of the widget is attached to the left side of the specified widget. If no widget ID is specified (for example, /Right), then the right side of the widget is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, Top=wid), then the top of the widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, /Top), then the top of the widget is attached to the top of the parent widget.

**Get/Set Value**

**getvalue** — Gets the Boolean array (0...rows–1, 0...cols–1) of selected cells. For information on cell selection, see the *Discussion* section.

**setvalue** — A two-element vector [row, column] specifying the cell to select.
Callback Parameters

Any table widget callback procedure must have the following three parameters:

- **wid** — The table widget ID.
- **cell** — A two-element vector \([row, column]\) specifying the modified cell.
- **value** — A string containing the new contents of the modified cell.

Discussion

Part of the resource specification for labeling the columns and rows can be specified using the `Name` keyword, otherwise the defaults are the *table.columnLabel and the *table.rowLabel resources, respectively (where columnLabel and rowLabel are the attributes).

**TIP** The `Clabels` and `Rlabels` keywords provide a method for “hard-coding” the column and row labels (respectively) in the application. For greater flexibility, create a resource file containing the column and row labels using a text editor, and load the resource specifications using WtResource. The `Name` keyword can then be used in the WwTable calling sequence to specify the resources.

Selecting Cells for Editing

When a cell is selected for editing, it is considered the current cell. You can delete and add characters in the current cell using the keys on your keyboard.

- To edit a cell, select it by clicking the left mouse button in the cell.
- To edit the cell to the left of the current cell, press the `<Shift>` and `<Tab>` keys at the same time.
- To edit the cell to the right of the current cell, press the `<Tab>` key.
- To edit the cell above the current cell, press the up arrow key.
- To edit the cell on the bottom of the current cell, press the down arrow key.

Selecting One or More Cells

This section describes how to select cells for operations other than editing.

- To select a single cell, click the middle mouse button on the cell.
- To extend the selection, press the `<Shift>` key and click the middle mouse button.
• To deselect the cell, press the <Shift> and <Control> keys at the same time and click the middle mouse button.

• To toggle the selection press the <Control> key and click the middle mouse button.

• To select a rectangular region of cells, press the middle mouse button, drag the mouse, and then release the middle mouse button.

• To select, deselect, extend, or toggle the selection of an entire row or column, click the middle mouse button (with appropriate keys listed above pressed) on the row or column label.

Example 1

The following example displays data stored in the phone_data variable in the save file:

WAVE_DIR/data/phone_example.sav

To restore this file, enter the following commands:

.RUN

RESTORE, !Data_Dir + ‘phone_example.sav’

COMMON Tablecomm, phone_data

END

Callback Procedures

PRO tableCB, wid, which, text

COMMON Tablecomm, phone_data

PRINT, ‘Table’, which, text

status = WwSetValue(wid, which)

PRINT, WwGetValue(wid)

END

PRO ButtonCB, wid, data

COMMON Tablecomm, phone_data

PRINT, ‘Table Selected’

shell = WwMainWindow(wid, form, /Vertical, $

TITLE = ‘Table’)

clrs = INTARR(N_TAGS(phone_data), $

N_ELEMENTS(phone_data))

FOR i = 0, N_ELEMENTS(phone_data)-1 DO $

clrs(*, i) = INDGEN(N_TAGS(phone_data)) * 20 $

rlabs = STRTRIM(STRING(INDGEN(N_ELEMENTS(phone_data))), 2) $

table = WwTable(form,’tableCB’,phone_data, $

$}
Colors=clrs, Visible=[10, N_TAGS(phone_data)-4], $ Clabels=TAG_NAMES(phone_data), Rlabels = rlabs, $ /Vertical)
status = WwSetValue(shell, /Display)

**Example 2**

This example demonstrates the use of the `Setcells` and `Setcelldata` keywords. In this program fragment, the contents of table cells are set to the value of the row number times the column number.

**Callback Procedures**

FUNCTION settableCLBK, wid, data, n, r, c
    PRINT, data
    RETURN, STRTRIM(STRING(r * c), 2)
    ; Return the string value of the product of row times column.
END

PROC buttonCB, wid, data
    COMMON Tablecomm, phone_data
    PRINT, 'Table Selected'
    shell = WwMainWindow(wid, form, /Vertical, $ TITLE = 'Table')
    table = WwTable(form, 'tableCLBK', Visible=[10, 5], $ Rows=40, Cols=50, Setcells = 'settableCLBK', $ Setcelldata='any var')
    status = WwSetValue(shell, /Display)
END

**Widget Commands**

top = WwInit('tble_ex', 'Examples', layout)
label = ['Table']
button = WwButtonBox(layout, label, 'ButtonCB')
status = WwSetValue(top, /Display)
WwLoop
button = WwButtonBox(layout, label, 'buttonCB')
status = WwSetValue(top, /Display)
WwLoop

See Also
WtTable, WwTableUtils

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, Using WAVE Widgets.

For additional information on the color and font keywords, see Setting Colors and Fonts on page 201.

For additional information on attachment keywords, see Form Layout: Attachments on page 170.

For information on Get and Set values, see Setting and Getting Widget Values on page 206.

WwTableUtils Function

Manages the contents of a table widget.

Usage

status = WwTableUtils(wid [, param1, ..., param9])

Input Parameters

wid — The widget ID of a WwTable widget.

param1, ..., param9 — (optional) These parameters depend on keyword use in the function calling sequence. See Keywords for more information.

Returned Value

status — A value indicating success or failure of the WwTableUtils call.

  1 Indicates success.

  0 Indicates failure.
Keywords

Add — Add specified rows or columns to the table at the specified position. This keyword must be used with either the Columns or the Rows modifier keywords.

If the Columns modifier keyword is used, you must specify all of the following parameters:

- param1 — The column before which new columns are added.
- param2 — A 2D string array of column values.
- param3 — The number of columns to be added.
- param4 — A 1D array of column widths.
- param5 — A 1D array of column labels.
- param6 — A 1D array of column maximum lengths.
- param7 — A 1D array of column alignments. Valid values are:
  - 0  Align cell contents to cell’s left edge (left justify).
  - 1  Center cell contents (center justify).
  - 2  Align cell contents to cell’s right edge (right justify).
- param8 — A 1D array of column label alignments. Valid values are:
  - 0  Align cell contents to cell’s left edge (left justify).
  - 1  Center cell contents (center justify).
  - 2  Align cell contents to cell’s right edge (right justify).
- param9 — A 2D array of column cell colors.

If the Rows modifier keyword is used, you must specify all of the following parameters:

- param1 — The row before which new rows are added.
- param2 — A 2D string array of row values.
- param3 — The number of rows to be added.
- param4 — A 1D array of row labels.
- param5 — A 2D array of row cell colors.
**CancelEdit** — If nonzero, cancels the edit made to the currently edited cell. (See the Edit keyword.)

**CommitEdit** — If nonzero, commits the edit made to the currently edited cell. (See the Edit keyword.)

**Delete** — Deletes specified rows or columns from the table at the specified position. This keyword must be used with either the Columns or the Rows modifier keywords. If you use the Delete keyword, you must also specify the following parameters:

- `param1` — (long) The position (row or column) in the table to begin deleting.
- `param2` — The number of rows or columns to delete.

**Deselect** — Deselects specified rows or columns. If the All modifier keyword is used with Deselect, all rows and columns are deselected. If used with the Rows or Columns modifier keywords, the specified rows or columns are deselected, and the following parameter must be specified:

- `param1` — Index of row or column to deselect.

**Edit** — If nonzero, begin editing at the specified row and column. If used with the Cell modifier keyword, begin editing at the specified cell. If you use the Edit keyword, you must also specify the following parameters:

- `param1` — Row of the cell to edit.
- `param2` — Column of the cell to edit.

**Get** — If used with the Cell modifier keyword, WwTableUtils returns the value of the specified cell (not 1 or 0). The parameters `param1` and `param2` must be used:

- `param1` — Row of the cell to get.
- `param2` — Column of the cell to get.

If used with the Visible modifier keyword, returns the rectangle of a visible cell, and the following parameters must be used:

- `param1` — (output) The top row of the cell.
- `param2` — (output) The bottom row of the cell.
- `param3` — (output) The left column of the cell.
- `param4` — (output) The right column of the cell.
Make — Makes the specified cell visible. This keyword must be used with the Visible modifier keyword, and you must also use the following parameters:

\[ param1 \] — The row of the cell to make visible.
\[ param2 \] — The column of the cell to make visible.

Redraw — Redraws the specified cell. This keyword must be used with the Cell modifier keyword, and you must also use the following parameters:

\[ param1 \] — The row of the cell to redraw.
\[ param2 \] — The column of the cell to redraw.

Select — If used with the Columns or Rows modifier keywords, selects the specified rows or columns, and the following parameter must be used:

\[ param1 \] — The index of the row or column to select.

If used with the Cell modifier keyword, selects specified cells, and the following parameters must be specified:

\[ param1 \] — The row of the cell to select.
\[ param2 \] — The column of the cell to select.

Set — Sets the value or color of a cell or column. This keyword must be used with the Cell, or Color and Rows, or Color and Columns modifier keywords.

If you use Set with the Cell modifier keyword, you must specify the following parameters:

\[ param1 \] — The row of the cell to set.
\[ param2 \] — The column of the cell to set.
\[ param3 \] — The new value of the specified cell.

If you use Set with the Color and Rows modifier keywords, you must specify the following parameters:

\[ param1 \] — The row at which to start setting colors.
\[ param2 \] — A 2D array of new color table indices for rows.
\[ param3 \] — The number of colors in the Color array.

If you use Set with the Color and Columns modifier keywords, you must specify the following parameters:
param1 — The column at which to start setting colors.

param2 — A 2D array of new color map indices for columns.

param3 — The number of elements in the Color array.

**Modifier Keywords**

**NOTE** These modifier keywords must be used in conjunction with the keywords listed.

*All* — When nonzero, deselects all cells in the table.

*Cell* — When nonzero, selects, deselects, edits, gets, or sets the color of the specified cell.

*Columns* — When nonzero, adds, deletes, deselects, or selects columns from the table.

*Color* — When nonzero, sets cell or column colors.

*Rows* — When nonzero, adds, deletes, deselects, or selects rows from the table.

*Visible* — When nonzero, obtains visible cells or makes cells visible.

**Discussion**

This function facilitates the portability between Microsoft Windows and X Windows applications.

**Examples**

The following ten code fragments demonstrate various uses of the WwTableUtils function.

(1) Add rows to a table

```lisp
new_strings = STRTRIM(STRING(LONARR(numcols, numrows)), 2)
new_rlabels = STRTRIM(STRING(LINDGEN(numrows)+start_row), 2)
status = WwTableUtils (tbl_wid, $ start_row, $ ; add after this row new_strings, $ ; use '0' strings numrows, $ ; number of new rows new_rlabels, $ ; new row labels /Add, /Rows)
```
(2) Delete rows from a table

```plaintext
status = WwTableUtils (tbl_wid, $
  start_row, $ ; start location
  numrows, $ ; number of rows to delete
 /Delete, /Rows)
```

(3) Add columns to a table

```plaintext
new_strings = STRTRIM(STRING(LONARR(numcols, numrows)), 2)
new_clabels = STRTRIM(STRING(LINDGEN(numcols)+ $
  start_column), 2)
new_colwidths = LONARR(numcols)+10
status = WwTableUtils (tbl_wid, $
  start_column, $ ; add after this column
  new_strings, $ ; use '0' strings
  numcols, $ ; number of new columns
  new_colwidths, $ ; new column widths
  new_clabels, $ ; new column labels
 /Add, /Columns)
```

(4) Delete columns from a table

```plaintext
status = WwTableUtils (tbl_wid, $
  start_column, $ ; start location
  numcols, $ ; number of columns to delete
 /Delete, /Columns)
```

(5) Get visible cells

```plaintext
status = WwTableUtils (tbl_wid, top_row, bottom_row, $
  left_column, $
  right_column, $
 /Get, /Visible)
```

(6) Set the value of an individual cell

```plaintext
status = WwTableUtils (tbl_wid, row, col, value, $
 /Set, /Cell)
```

(7) Redraw a cell

```plaintext
status = WwTableUtils (tbl_wid, row, col, /Redraw, /Cell)
```
(8) Deselect all cells

status = WwTableUtils (tbl_wid, /Deselect, /All)

(9) Select a cell

status = WwTableUtils (tbl_wid, row, col, /Select, /Cell)

(10) Make a cell visible

status = WwTableUtils (tbl_wid, row, col, /Make, /Visible)

See Also
WwTable

---

**WwText Function**

Creates a text widget that can be used for both single-line text entry or as a full text editor. In addition, this function can create a static text label.

**Usage**

\[text = WwText(parent, verifyCallback)\]

**Input Parameters**

*parent* — The widget ID of the parent widget.

*verifyCallback* — (Optional) A string containing the name of a callback routine that is executed when the contents of the text field changes.

**Returned Value**

*text* — The text widget ID.

**Keywords**

*Cols* — Specifies the number of columns in the text field (in characters).

*File* — Specifies a name of a file containing text.
HScroll — (Windows only) If nonzero, multiline text windows are created with a horizontal scroll bar.

Label — If nonzero, the widget is a label (static text). The value of the label is set using the Text keyword. If Label is nonzero and Text is not used, the function looks for a label in a resource specification (see Discussion).

If the Label keyword is set to a string value, the text field (single line) widget or text edit (multi-line) widget is preceded by this string.

Layout_name — A string containing the name of the Form widget created when Form is specified and both Text and Label are specified as strings. The Layout_name specified is part of the resource specification. (Default: caption)

Name — Specifies the name of the Label and/or Text widgets. The Name specified is part of the resource specification containing the label or text widget names. The default for Name depends on other keywords being specified and how they are specified in the WwText usage.

    If both Label and Text are specified as null strings, Name is a two-element string array naming the Text widget and the Label widget (in that order). (Default: *caption.text.value; and *caption.label.labelString, respectively)

Pixmap — Specifies a filename containing a pixmap or bitmap file to be displayed as a label. This keyword is used in conjunction with the Label keyword.

Position — If the text widget is to be placed in a bulletin board layout, use this keyword to specify the x, y coordinates of the text widget within the bulletin board.

Read — If present and nonzero, the text widget is read only.

Rows — Specifies the number of rows in the text field.

Text — Specifies a label for the text field. If Text is not defined or is defined as a null string, the function looks for the text field label in a resource specification (see Discussion).

VScroll — (Windows only) If nonzero, multiline text windows are created with a vertical scroll bar.

Color/Font Keywords

Background — Specifies the background color name.

Font — Specifies the name of the font used for text.

Foreground — Specifies the foreground color name.
Attachment Keywords

**Bottom** — If a widget ID is specified (for example, Bottom=wid), then the bottom of the text widget is attached to the top of the specified widget. If no widget ID is specified (for example, /Bottom), then the bottom of the text widget is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, Left=wid), then the left side of the text widget is attached to the right side of the specified widget. If no widget ID is specified (for example, /Left), then the left side of the text widget is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, Right=wid), then the right side of the text widget is attached to the left side of the specified widget. If no widget ID is specified (for example, /Right), then the right side of the text widget is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, Top=wid), then the top of the text widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, /Top), then the top of the text widget is attached to the top of the parent widget.

Get/Set Value

getvalue — A string containing the displayed text.

setvalue — A string containing the displayed text.

Callback Parameters

Any text widget callback procedure must have the following two parameters:

**wid** — Text widget ID.

**parent** — Parent widget ID.

Discussion

If the **Label** keyword is set to a null string and **Name** is not used, the function looks in *caption.label.labelString for the label resource. The text resource can also be specified using the **Name** keyword if the **Text** keyword is specified as a null string. The text resource default is *caption.text.labelString.

Windows USERS  By default, if the linelength or number of lines of text in the widget exceeds the size of the widget, horizontal and/or vertical scroll bars are
attached to the widget automatically. To force the attachment of scroll bars to a text widget, use the *HScroll* and/or *VScroll* keywords. When these keywords are used, scroll bars will not dynamically appear or disappear once the text widget is created.

---

**TIP** The *Label* and *Text* keywords provide a method for “hard-coding” labels and text field labels (respectively) in the application. For greater flexibility, define a resource file containing the label or text by using WtResource, and use the *Name* keyword in the WwText calling sequence to access the resources.

---

**Example 1: Single-line Text Field and Label**

This example creates a layout widget containing a single-line text field and a label. The callback is executed when the user enters text in the text field and presses <Return>.

Enter the callback procedure into a file, and compile the procedure with the .RUN command. Then, enter the widget commands at the **WAVE** prompt (or enter them in a command file and run it with the @ command). To dismiss the button box, select the appropriate function (such as *Close*) from the window manager menu.

**Callback Procedure**

```wave
PRO TextCB, wid, data
    PRINT, 'Text done'
    value = WwGetValue(wid)
    PRINT, value
END
```

**Widget Commands**

```wavedoc
top=WwInit('ww_ex35', 'Examples', layout, /Form)
; Initialize WAVE Widgets and create the form layout widget.
label = WwText(layout, /Label, Text='This is Label')
; Create the label widget.
text = WwText(layout, 'TextCB', Cols=40, left=label)
; Create the single-line text field widget, attaching it to the right edge of
; the label widget.
status = WwSetValue(top, /Display)
WwLoop
```

**WwText Function** 355
Example 2: Multi-line Text Window

This example creates a multi-line text window. Because the *Read* keyword is used, the text is read-only.

**Callback Procedure**

```plaintext
PRO TextCB, wid, data
    PRINT, 'Text done'
    value = WwGetValue(wid)
    PRINT, value
END
```

**Widget Commands**

```plaintext
top = WwInit('ww_ex36', 'Examples', layout)
filename = GETENV('WAVE_DIR')+'/Tips'
text = WwText(layout, 'TextCB', /Read, $
    File = filename, Cols=40, Rows=20)
status = WwSetValue(top, /Display)
WwLoop
```

**See Also**

WgTextTool (see the PV-WAVE Reference)

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, *Using WAVE Widgets*.

For additional information on the color and font keywords, see *Setting Colors and Fonts* on page 201.

For additional information on attachment keywords, see *Form Layout: Attachments* on page 170.

For information on Get and Set values, see *Setting and Getting Widget Values* on page 206.
**WwTimer Function**

Registers a WAVE Widgets timer procedure.

**Usage**

\[ status = \text{WwTimer}(time, \text{timer\_proc} [, \text{userdata}]) \]

**Input Parameters**

- **time** — Specifies the time interval in milliseconds.
- **timer\_proc** — Specifies the name of a PV-WAVE timer procedure.
- **userdata** — (optional) User-defined data.

**Returned Value**

- **status** — A value indicating success or failure of the timer registration.
  
  1 Indicates success.
  
  0 Indicates failure.

**Keywords**

None.

**Timer Procedure Parameters**

The following parameters are required for the timer procedure:

- **wid** — The ID of the top-level widget.
- **userdata** — User-defined data.

**Discussion**

This procedure relies on the WtTimer procedure to add a timer in *Once\_Only* mode (the timer procedure is executed only once after the specified time interval).
Example

This example comes from the code for the WzAnimate VDA Tool. The procedure shown in this example is used by the WzAnimate VDA Tool to restart the timer.

```vwa
PRO WzAnimateStartTimer, tool_name
 ; Forward declaration of function.
 DECLARE FUNC, TmGetAttribute
 DECLARE FUNC, TmSetAttribute
 delay = TmGetAttribute(tool_name, ‘TM’, ‘DELAY’)
 timerid = WwTimer(delay, ‘WzAnimateTimerCB’, tool_name)
 tmp = TmSetAttribute(tool_name, ‘TM’, ‘TIMER’, 1B)
END
```

See Also

WtTimer

---

**WwToolBox Function**

Creates an array of graphic buttons (icons).

**Usage**

```vwa
toolb = WwToolBox(parent, labels, callback)
```

**Input Parameters**

- **parent** — The widget ID of the parent widget.
- **labels** — A string array of icon file names.
- **callback** — A string containing the name of the callback routine that is executed when a button is selected.

**Returned Value**

- **toolb** — The ID of the tool box widget. If only one button is requested, that button’s widget ID is returned.
Input Keywords

**Border** — Specifies the width in pixels of the tool box and icon borders. The default is 0.

**Center** — An array specifying the position of the left and right edge of buttons as a percentage of tool box width. By default, buttons are spaced evenly in the box.

**Form** — When present and nonzero, buttons are placed in a form layout and all specified attachment keywords are honored (i.e., /Left, /Right, /Top, /Bottom). By default, buttons are placed in a row/column layout.

**Horizontal** — When present and nonzero, creates a horizontally aligned row of buttons (the default).

**Layout_name** — Names the Form widget (if Form is specified), or the RowColumn widget (if Horizontal or Vertical is specified). (Default: tools)

**Measure** — If you create a horizontal box, then measure specifies the number of rows into which the buttons will be divided. For a vertical box, this keyword specifies the number of columns in which to divide the buttons.

**Nofmany** — Used in conjunction with the Radio keyword. If nonzero, creates non-exclusive radio buttons where any number of buttons can be selected at once.

**Oneofmany** — Used in conjunction with the Radio keyword. If nonzero, creates exclusive radio buttons where only one button can be selected at a time. (Refer to the note regarding the callback input parameter in the Discussion section for more information on using Oneofmany.)

**Position** — If the tool box widget is to be placed in a bulletin board layout, use this keyword to specify the x, y coordinates of the tool box widget within the bulletin board.

**Radio** — If nonzero, uses radio (toggle) buttons. (Default: push buttons.)

**ShadowThickness** — (UNIX only) If specified and nonzero, sets the shadow thickness of the button in pixels. (Default: 2)

**Spacing** — Specifies the space in pixels between buttons. The default is 0.

**Vertical** — When present and nonzero, creates a vertically aligned column of buttons.

Output Keywords

**Tools** — Returns an array of graphical button (icon) widget IDs.
**Color/Font Keywords**

*Background* — Specifies the background color name.

*Basecolor* — Specifies the base color.

*Font* — Specifies the name of the font used for button text.

*Foreground* — Specifies the foreground color name.

**Attachment Keywords**

*Bottom* — If a widget ID is specified (for example, `Bottom=wid`), then the bottom of the tool box widget is attached to the top of the specified widget. If no widget ID is specified (for example, `/Bottom`), then the bottom of the tool box widget is attached to the bottom of the parent widget.

*Left* — If a widget ID is specified (for example, `Left=wid`), then the left side of the tool box widget is attached to the right side of the specified widget. If no widget ID is specified (for example, `/Left`), then the left side of the tool box widget is attached to the left side of the parent widget.

*Right* — If a widget ID is specified (for example, `Right=wid`), then the right side of the tool box widget is attached to the left side of the specified widget. If no widget ID is specified (for example, `/Right`), then the right side of the tool box widget is attached to the right side of the parent widget.

*Top* — If a widget ID is specified (for example, `Top=wid`), then the top of the tool box widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, `/Top`), then the top of the tool box widget is attached to the top of the parent widget.

**Get/Set Value**

*getvalue* — Icon (pixmap) ID.

*setvalue* — The icon file name.

**Callback Parameters**

Any toolbox callback procedure must have these parameters:

*`wid`* — Button widget ID.

*`index`* — Index of the button pushed (1 – n).
Discussion

This function creates a box containing iconic buttons arranged in rows and/or columns. If you only create a single button, then the box widget is not created — only the button widget is created.

NOTE  The callback procedure is called twice if the Oneofmany keyword is specified. It is called once when the previously selected button is deselected and again for the newly selected button. Use WwGetValue to determine if a button is currently selected or deselected.

Example

This example creates a box containing graphical buttons (a toolbox). The callback routine, PickedCB, is executed when one of the toolbox buttons is selected.

Enter the callback procedure into a file, and compile the procedure with the .RUN command. Then, enter the widget commands at the WAVE> prompt (or enter them in a command file and run them with the @ command). To dismiss the tool box, select the appropriate function (such as Close) from the window manager menu.

Callback Procedure

PRO PickedCB, wid, which
CASE which OF
  1: PRINT,‘Search Selected’
  2: PRINT,‘Toc Selected’
  3: PRINT,‘Topics Selected’
  4: PRINT,‘Quit Selected’
ENDCASE
END

Widget Commands

top = WwInit(‘ww_ex37’, ‘Examples’, layout, height=300, width=600)
pixmaps = [GETENV(‘WAVE_DIR’)+ $’/xres/wxpm_btn_help_search’,$
          GETENV(‘WAVE_DIR’)+ $’/xres/wxpm_btn_help_toc’,$
          GETENV(‘WAVE_DIR’)+ $’/xres/wxpm_btn_help_topics’,$
          GETENV(‘WAVE_DIR’)+ $’/xres/wxpm_btn_help_quit’]
dbox = WwToolBox(layout, pixmaps, ‘PickedCB’, /Vertical, $Spacing=20, Measure=2)
status = WwSetValue(top, /Display)
WwLoop

**Windows USERS**  In the preceding code, you must change the icon names as follows: change wxpm to wxbm and add the extension .bmp. For example, change wxpm_btn_help_search to wxbm_btn_help_search.bmp.

**See Also**

For more information about how to write an application program based on WAVE Widgets, refer to *Chapter 5, Using WAVE Widgets*.

For additional information on the color and font keywords, see *Setting Colors and Fonts* on page 201.

For additional information on attachment keywords, see *Form Layout: Attachments* on page 170.

For information on Get and Set values, see *Setting and Getting Widget Values* on page 206.
**Widget Toolbox Reference**

**WtAddCallback Function (Motif Only)**

Registers a callback routine for a given widget.

**Usage**

```
status = WtAddCallback(widget, reason, callback [, client_data])
```

**Input Parameters**

- **widget** — The widget ID of the widget to add the callback to (long).
- **reason** — A string containing the callback reason. This parameter is GUI-dependent. See the *Discussion* section below for more information.
- **callback** — A string containing the name of the callback routine.
- **client_data** — A variable. The value of this variable is passed to the callback routine.

**Returned Value**

- **status** — One (1) indicates success; zero (0) indicates failure.
Keywords

**Noparams** — If present and nonzero, the callback is called with two parameters: *wid* and *data*. All other parameters, as discussed in Appendix B, *Motif Callback Parameters*.

Discussion

**Windows USERS**  The use of WtAddCallback is not supported for Windows.

Callback reasons are listed throughout the *OSF/Motif Programmer’s Reference*. To use a callback reason in PV-WAVE, remove the XmN or XtN prefix. For example:

<table>
<thead>
<tr>
<th>Motif Reason</th>
<th>WAVE Widget Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>XmNactivateCallback</td>
<td>activateCallback</td>
</tr>
</tbody>
</table>

The application can optionally use the *client_data* parameter to specify some application-defined data to be passed to the callback procedure when the callback is invoked. If *client_data* is a local variable (defined only in the current procedure), a copy of that variable is created and passed (passed by value). If the *client_data* is a global variable (defined in a Common Block), it is passed by reference.

Example

This example creates a Motif button labeled **Done**. When you select the button, the widget is destroyed. To run the example, enter the callback and the example procedures in a file and run them with .RUN.

**Callback Procedure Example: Motif**

This is the callback routine. Note that the callback routine for the pushbutton widget class requires six parameters. The required callback parameters for Motif widget classes are discussed in *Motif Widget Classes* on page A-1.

```pro
PRO CancelHelp, wid, data, npar, reason, $
  event, count
  COMMON block, top
  status=WtClose(top)
END
```
**Example Procedure**

```plaintext
PRO example
  COMMON block, top
  @wtxmclasses.pro
  top=WtInit('wt_ex1', 'Examples')
  widget=WtCreate('Done', xmPushButtonWidgetClass, top)
  status=WtAddCallback(widget, 'activateCallback', 'CancelHelp')
  status=WtSet(top, /Realize)
  WtLoop
END
```

**See Also**

`WwCallback`

---

**WtAddHandler Function**

Registers the event handler function for a given widget.

**Usage**

```plaintext
status = WtAddHandler(widget, eventmask, handler [, client_data])
```

**Input Parameters**

- `widget` — The ID of the widget to add the event handler to (long).
- `handler` — A string containing the X event handler procedure name.
- `client_data` — A variable. The value of this variable is passed to the callback routine.

**Returned Value**

- `status` — One (1) indicates success; zero (0) indicates failure.
Keywords

*Nonmaskable* — If present and nonzero, nonmaskable events are intercepted by the event handler. Such events include GraphicsExposure, NoExposure, Selection-Clear, SelectionRequest, SelectionNotify, ClientMessage, and MappingNotify.

*Noparams* — If nonzero, calls the event handler procedure with three parameters: `widget` (the widget ID), `client_data`, and the `event` structure.

Discussion

**Windows USERS** Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

An event handler is a PV-WAVE procedure that is executed when a specific type of event occurs within a widget. Some, all, or no X events can be handled using one or more event handlers.

Under Windows, Windows events are mapped to X events as follows:

<table>
<thead>
<tr>
<th>Windows Events</th>
<th>X Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>WM_KEYDOWN, WM_CHAR, WM_SYSCHAR</td>
<td>KeyPressMask</td>
</tr>
<tr>
<td>WM_KEYUP</td>
<td>KeyReleaseMask</td>
</tr>
<tr>
<td>WM_KILLFOCUS, WM_SETFOCUS</td>
<td>FocusChangeMask (FocusOut, FocusIn)</td>
</tr>
<tr>
<td>WM_LBUTTONDOWN,</td>
<td>ButtonPressMask</td>
</tr>
<tr>
<td>WM_MBUTTONDOWN,</td>
<td></td>
</tr>
<tr>
<td>WM_RBUTTONDOWN</td>
<td>ButtonReleaseMask</td>
</tr>
<tr>
<td>WM_LBUTTONUP,</td>
<td></td>
</tr>
<tr>
<td>WM_MBUTTONUP,</td>
<td></td>
</tr>
<tr>
<td>WM_RBUTTONUP</td>
<td></td>
</tr>
<tr>
<td>WM_NCHITTEST,</td>
<td>PointerMotionMask</td>
</tr>
<tr>
<td>WM_MOUSEMOVE</td>
<td></td>
</tr>
<tr>
<td>WM_PAINT</td>
<td>ExposureMask</td>
</tr>
</tbody>
</table>
Event Structure for Windows Platforms

The event structure returned by the event handler (or callback) procedure on Windows platforms contains the following tag fields:

- **event** — Event handle.
- **type** — Event type (ButtonPress, KeyPress etc., as listed in the file wtxlib.pro in the Standard Library).
- **hwnd** — Windows window handle.
- **message** — Windows message ID.
- **wParam** — Windows WPARAM.
- **lParam** — Windows LPARAM.
- **time** — Time of the event.
- **x** — x-coordinate of the mouse pointer.
- **y** — y-coordinate of the mouse pointer.
- **state** — Indicates the state of all of the buttons and modifier keys at the time of the event, represented by a mask of the button and modifier key symbols.
- **button** — Indicates which button changed state to trigger this event.

For information on the requirements for writing event handler procedures, see Adding Event Handlers on page 223.

Example

The following code fragments demonstrate the use of WtAddHandler.

```c
.
.
pane=WtCreate('menu', PopupMenuWidget, parent)
status = WtAddHandler(pane, ButtonPressMask, '
  'PostMenu', parent)
```
PRO PostMenu, wid, parent, nparams, mask, event
    @wtxlib
    status=WtPointer("GetLocation", wid,state)
    if (Button3Mask AND state(6)) ne 0 then $
    status=WtSet(pane,POPUP=event)
END

X Event Handler Procedure Example
PRO handler, widget, data, nparams, mask, event
    ...
END

See Also
For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to Chapter 6, Using the Widget Toolbox.

WtClose Function
Closes the current Xt session, and destroys all children of the top-level widget created in WtInit. This routine can also be used to destroy additional widget trees.

Usage
status = WtClose(widget)

Parameters
widget — The widget ID of the top-level shell (long).

Returned Value
status — One (1) indicates success; zero (0) indicates failure.

Keywords
None.
Discussion

Windows USERS Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

This function is usually called in a callback routine to destroy a popup shell (dialog, etc.).

Example

This example creates a Motif button labeled Done. When you select the button, the widget is destroyed. To run the example, enter the callback and the example procedures in a file and run them with .RUN.

Callback Procedure Example: Motif

This is the callback routine. Note that the callback routine for the pushbutton widget class requires six parameters. The required callback parameters for Motif widget classes are discussed in Appendix B, Motif Callback Parameters.

```
PRO CancelHelp, wid, data, npar, reason, event, count
    COMMON block, top
    status=WtClose(top)
END
```

Example Procedure

```
PRO example
    common block, top
    @wtxmclasses.pro
    top=WtInit(‘wt_ex2’, ‘Examples’)
    widget=WtCreate(‘Done’, xmPushButtonWidgetClass, top)
    status=WtAddCallback(widget, ‘activateCallback’, ‘CancelHelp’)
    status=WtSet(top, /Realize)
    WtLoop
END
```
See Also
For more information about how to write an application program based on the PWAVE Widget Toolbox, refer to Chapter 6, Using the Widget Toolbox.

WtCreate Function (Motif Only)

Creates a widget or shell instance specified by widget class.

Usage

\[ \text{widget} = \text{WtCreate}(\text{name, class, parent [, argv]} ) \]

Parameters

name — A string containing the name of the widget or shell to be created.

class — A constant value specifying the widget or shell class (long). See Appendix A, Motif Widget Classes for a list of Motif widget classes.

parent — The widget or shell ID of the parent (long).

argv — A structure that contains the arguments or resources for the widget or shell.

Returned Value

widget — A newly created widget. If the function fails, zero (0) is returned.

Keywords

ConfirmClose — A string containing the name of the procedure called when the user selects the Close or Quit menu button from the window manager menu.

UserData — A variable. If the ConfirmClose keyword is specified, the value of this variable is passed to the Close or Quit callback procedure.

Discussion

The ConfirmClose keyword lets you control what happens when the user selects Close or Quit from the window manager menu. Normally, the window from which the menu item was selected is destroyed; however, you might want to display a confirmation dialog box or take another action instead of simply allowing the window
to be destroyed. The callback procedure specified by *ConfirmClose* destroys the window when appropriate.

The *ConfirmClose* procedure you specify accepts two parameters: *wid* and *user_data*, where:

\[ wid = \text{The widget ID of the top-level shell of the application.} \]
\[ user\_data = \text{The variable specified via the *User\_Data* keyword. If *User\_Data* is not specified, 0 (zero) is passed to the *ConfirmClose* routine.} \]

If specified, your *ConfirmClose* routine must close the top-level shell of the application. An example of a simple *ConfirmClose* routine which just closes the shell is:

```
PRO MyConfirmClose, wid, user_data
    s = WwSetValue(wid, /Close)
END
```

If *ConfirmClose* is not specified, then the shell is simply closed.

**Example**

```
items = REPPLICATE({FLATNON, label: '', mnemonic: ''}, 3)
items(0) = {FLATNON, 'Bold', 'B'}
items(1) = {FLATNON, 'Italic', 'I'}
items(2) = {FLATNON, 'Underline', 'U'}
fargs = {items: items}
widget = WtCreate('fnon', fargs)
```

**NOTE**  All widgets are managed when created. To unmanage them after creation use *WtSet* (wid, /Unmanage).

**See Also**

*WtSet*

For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to *Chapter 6, Using the Widget Toolbox.*
### WtCursor Function

Sets or changes the cursor.

#### Usage

```c
status = WtCursor(function, widget [, index])
```

#### Parameters

- **function:**
  - `'Default'` — The default is the system cursor.
  - `'System'` — Sets the default system cursor.
  - `'Wait'` — Sets the wait cursor.
  - `'Set'` — Sets the specified cursor. If `'Set'` is specified, the `index` parameter follows:
    - `index` — The cursor index (e.g., XC_X_cursor). See Appendix C, Widget Toolbox Cursors for a list of cursors.

- **widget** — The ID of the widget for which the cursor is being set.

#### Returned Value

- **status** — One (1) indicates success; zero (0) indicates failure.

#### Discussion

**Windows USERS** Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

This routine changes the current cursor for a given widget to a new cursor defined by `index`. The following cursors are available:

- All XC_* cursor types (for Motif and Windows) are listed in Appendix C, Widget Toolbox Cursors. For additional information on these cursors, see

• A set of custom cursors designed by Visual Numerics listed in Appendix C, Widget Toolbox Cursors.

Example

This example demonstrates a callback called to display the heartbeat.dat file with the WgMovieTool procedure. Because it takes a while to read the data file into PV-WAVE, the wait cursor is set before the file is read to notify the user that the file is being read:

```pro
PRO MovieCB, wid, index
@wtcursor
top = WwGetValue(wid, /Userdata)
CASE index OF
1: BEGIN
status = WtCursor('WAIT', top)
heart = BYTARR(256, 256, 15)

IF !Version.platform EQ 'vms' THEN $
OPENR, u, getenv('WAVE_DIR')+ $'
    '[data]heartbeat.dat', /Get_Lun
ELSE
OPENR, u, !Data_Dir + 'heartbeat.dat', /Get_Lun
READU, u, heart
CLOSE, u
WgMovieTool, heart, top, movie, widx, 1, /Popup, /Do_tvscl
status = WtCursor('DEFAULT', top)
END
2: BEGIN
status = WwSetValue(top, /Close)
END
ENDCASE
END
```

See Also

For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to Chapter 6, Using the Widget Toolbox.
**WtGet Function**

Retrieves widget resources.

**Usage**

\[ value = \text{WtGet}(\text{widget} [, \text{resource}]) \]

**Parameters**

*widget* — The widget ID.

*resource* — (optional, Motif only) A string containing the name of the requested resource. This parameter is GUI-dependent. See the Discussion section for more information.

**Returned Value**

*value* — A variable in which the value of the resource is returned. The data type of *value* depends on the requested resource.

**Keywords**

*Child=child* — (Motif only) Returns the ID of the child widget in a composite widget, such as a Command or FileSelection widget:

Legal values for a Command widget:

- XmDIALOG_COMMAND_TEXT
- XmDIALOG_PROMPT_LABEL
- XmDIALOG_HISTORY_LIST

Legal values for a FileSelection widget:

- XmDIALOG_APPLY_BUTTON
- XmDIALOG_CANCEL_BUTTON
- XmDIALOG_DEFAULT_BUTTON
- XmDIALOG_DIR_LIST
- XmDIALOG_DIR_LIST_LABEL
- XmDIALOG_FILTER_LABEL
For a MessageBox widget:
- XmDIALOG_CANCEL_BUTTON
- XmDIALOG_DEFAULT_BUTTON
- XmDIALOG_HELP_BUTTON
- XmDIALOG_MESSAGE_LABEL
- XmDIALOG_OK_BUTTON
- XmDIALOG_SEPARATOR
- XmDIALOG_SYMBOL_LABEL

For a SelectionBox widget:
- XmDIALOG_APPLY_BUTTON
- XmDIALOG_CANCEL_BUTTON
- XmDIALOG_DEFAULT_BUTTON
- XmDIALOG_HELP_BUTTON
- XmDIALOG_LIST
- XmDIALOG_LIST_LABEL
- XmDIALOG_OK_BUTTON
- XmDIALOG_SEPARATOR
- XmDIALOG_TEXT
- XmDIALOG_WORK_AREA

**Child** — Returns a long integer array of child widget IDs.
**Class** — Returns the widget class for the given widget.

**Count=count** — Specifies the number of items for resources containing an array of strings, such as a list or command.

**Destroyed** — Returns 1 if the given widget is being destroyed; otherwise returns 0.

**Managed** — Returns 1 if the given widget is managed; returns 0 if the widget is unmanaged.

**MultiClick** — Returns the time, in milliseconds, that is used to determine if consecutive mouse button clicks are to be interpreted as multiple clicks.

**Name** — Returns the name of the given widget.

**Name=name** — Returns the ID of the named widget and its parent.

**Ncols** — Specifies the number of columns of the two-dimensional resource to be retrieved. A two-dimensional resource is a resource whose value is a two-dimensional array of strings. Currently, two-dimensional resources are used in the table widget.

**Nrows** — Specifies the number of rows of the two-dimensional resource to be retrieved.

**Parent** — Returns the widget ID of the parent of the given widget.

**Realized** — Returns 1 if the given widget is realized (displayed); returns 0 if it is unrealized.

**Sensitive** — Returns 1 if the given widget is sensitive; returns 0 if it is not sensitive.

**Shell** — Returns 1 if the given widget is a shell (top-level widget); returns 0 if it is not a shell.

**Userdata** — Returns the user data (any variable or structure) for the given widget formally stored by the WtSet function.

**Value** — Returns the value for a scale, scroll bar, or toggle button.

**Widget** — Returns 1 if the given widget is a widget (not a shell); returns 0 if it is not a widget.

**Window** — Returns the window ID of the given widget.

**Window=window** — Returns the widget ID of the given window.
Discussion

**Windows USERS** Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

**Windows USERS** The WtGet keywords are supported for Windows; however, the use of the *resource* parameter to pass resource names is not supported.

Motif developers, see the *OSF/Motif Programmer’s Reference* for a list of resource names.

The resource name for a Widget Toolbox widget is derived from the Motif widget set resource name. Remove the XmN prefix from the Motif resource name. For example:

<table>
<thead>
<tr>
<th>Motif</th>
<th>PV-WAVE Widget Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>XmNwidth</td>
<td>width</td>
</tr>
<tr>
<td>XmNx</td>
<td>x</td>
</tr>
</tbody>
</table>

The data type of a resource’s value depends on the type of the resource.

Example

```plaintext
window = WtGet(wid, /Window)
; Returns the window ID of the specified widget (wid).

wid = WtGet(wid, Window=win(2))
; Returns the widget ID of window win(2) for the specified
; widget (wid).

widget = WtGet(wid, /Widget)
; Returns 1 if the given widget is not a shell, or 0 if it is a shell.

name = WtGet(wid, /Name)
; Returns the name of the specified widget (wid).

child = WtGet(wid, Child=XmDIALOG_DIR_LIST)
; Returns the ID of the XMDIALOG_DIR_LIST component of the
; FileSelection widget (wid).
```
See Also
For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to Chapter 6, Using the Widget Toolbox.

WtInit Function
Initializes the Widget Toolbox and the Xt toolkit, opens the display, and creates the first top-level shell.

Usage

\[ \text{topshell} = \text{WtInit}(\text{app\_name}, \text{appclass\_name} [, \text{Xserverargs} ...]) \]

Parameters

- **app\_name** — A string containing the name of the application as used in the resource file.
- **appclass\_name** — A string containing the application class name (name of the resource file).
- **Xserverargs** — (optional) A string containing X server arguments (font, display, synchronize, etc.). See the OSF/Motif Programmer’s Reference for more information.

Returned Value

- **topshell** — Returns the widget ID of the top application shell.

Keywords

- **Colors** — (Motif only) The maximum number of color table indices to be used. Otherwise, PV-WAVE uses all of the available color indices. On Windows platforms, PV-WAVE uses all available color indices.
- **ConfirmClose** — A string containing the name of the procedure called when the user selects the Close or Quit menu button from the window manager menu.
- **Context** — Specifies application context handle created in an Xt Intrinsics-based application statically linked with PV-WAVE. If this keyword is specified, the Top-shell keyword is also required.
**Resource** — Specifies a resource file to load into the resource manager database. The resources are used to set attributes of the top-level shell created by WtInit. The defaults are as follows:

(UNIX)  
WAVE_DIR/bin/Wave.ad

(OpenVMS)  
WAVE_DIR:[BIN]WAVE.AD

(Windows)  
WAVE_DIR\bin\wave.ad

Where WAVE_DIR is the main PV-WAVE directory.

**Topshell** — Specifies top-level shell handle of the main application shell created in an Xt Intrinsics-based application statically linked with PV-WAVE. If this keyword is specified, the Context keyword is also required.

**Windows USERS** The Context and Topshell keywords are not supported on Windows platforms.

**UserData** — A variable. If the ConfirmClose keyword is specified, the value of this variable is passed to the Close or Quit callback.

**Discussion**

**Windows USERS** Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

Call this routine before the first use of any Widget Toolbox routines, or to reinitialize Widget Toolbox after closing the top-level shells.

The ConfirmClose keyword lets you control what happens when the user selects Close or Quit from the window manager menu. Normally, the window from which the menu item was selected is destroyed; however, you might want to display a confirmation dialog box or take another action instead of simply allowing the window to be destroyed. The callback procedure specified by ConfirmClose destroys the window when appropriate.

The ConfirmClose procedure you specify accepts two parameters: wid and user_data, where:
\( wid \) = The widget ID of the top-level shell of the application.

\( user\_data \) = The variable specified via the \texttt{User\_Data} keyword. If \texttt{User\_Data} is not specified, 0 (zero) is passed to the \texttt{ConfirmClose} routine. If specified, your \texttt{ConfirmClose} routine must close the top-level shell of the application. An example of a simple \texttt{ConfirmClose} routine which just closes the shell is:

```pro
PRO MyConfirmClose, wid, user_data
    s = WwSetValue(wid, /Close)
END
```

If \texttt{ConfirmClose} is not specified, then the shell is simply closed.

**Example**

This example creates a Motif button labeled \textbf{Done}. When you select the button, the widget is destroyed. To run the example, enter the callback and the example procedures in a file and run them with .RUN.

**Callback Procedure Example: Motif**

```pro
PRO CancelHelp, wid, data, npar, reason, event, count
    COMMON block, top
    status=WtClose(top)
END
```

**Example Procedure**

```pro
PRO example
    COMMON block, top
    @wtxmclasses.pro
    top=WtInit('wt_ex3', 'Examples')
    widget=WtCreate('Done', xmPushButtonWidgetClass, top)
    status=WtAddCallback(widget, $
        'activateCallback', 'CancelHelp')
    status=WtSet(top, /Realize)
    WtLoop
END
```
See Also
For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to Chapter 6, Using the Widget Toolbox.

WtInput Function (Motif Only)

Registers an input source handler procedure.

Usage

\[ status = \text{WtInput}(\text{function}[, \text{parameters}]) \]

Input Parameters

function:

‘Add’ — Add input to the source of events. If this function is specified, provide the following parameters:

- \text{file_lun} — LUN of the input source file (or pipe).
- \text{handler} — (optional) A PV-WAVE procedure that is called when input is available.
- \text{client_data} — (optional) User data to be passed to the handler procedure.

‘Remove’ — Remove a previously registered input source. If this function is specified, provide the following parameter:

- \text{input_id} — ID of the input source being removed.

Returned Value

For ‘Add’:

\text{status} — The input source ID, or zero (0) to indicate failure.

For ‘Remove’:

\text{status} — One (1) indicates success; zero (0) indicates failure.

Keywords

\text{Noparams} — If nonzero, calls the input handler procedure with two parameters: top (the top-level shell of the application), and \text{client_data}.
If the ‘Add’ function is specified, the following keywords can be used:

**Except** — If specified and nonzero, the Xt Intrinsic condition XtInputExceptMask is used to define the input source.

**Read** — If specified and nonzero, the Xt Intrinsic condition XtInputReadMask is used to define the input source (default).

**Write** — If specified and nonzero, the Xt Intrinsic condition XtInputWriteMask is used to define the input.

### Discussion

**NOTE** This function is not available for Windows.

While most GUI applications are driven only by events, some applications need to incorporate other sources of input into the X Toolkit event handling mechanism. WtInput supports input or output gathering from files. The application registers an input source handler procedure and a file with the X Toolkit. When input is pending on the file, the registered handler is invoked.

**NOTE** In this context a “file” should be loosely interpreted to mean any sink (destination of output) or pipe (source of data).

For information on the requirements for writing input handler procedures, see *Adding Input Handler Procedures (Motif Only)* on page 226.

### Example

The following application accepts and processes data from another application that gathers the data.

```pro
PRO Server, top, client_data, nparams, id, lun, source
  READU, lun, data
    ; Process received data here.
END

PRO ButtonCB, wid, index
  COMMON ProcessComm, top, inputid
    ; Handle buttons here.
  CASE index OF
    1: ; Store
    2: ; Display
```
3: BEGIN ; Close application
status = WtInput('REMOVE', inputid)
status = WwSetValue(top, /Close)
END
ENDCASE
END

PRO ProcessData
    COMMON ProcessComm, top, inputid
    ; Spawn data gathering application
    SPAWN, 'getdata', unit = lun
    ; Initialize the application.
top = WwInit('processdata','Test', layout)
buttons = WwButtonBox(layout, $
    ['Store','Display','Close'], 'ButtonCB')
    ; Register input source handler and start the application.
inputid = WtInput('ADD', lun, 'Server')
status = WwSetValue(top, /Display)
WwLoop
    FREE_LUN, lun
END

See Also

For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to Chapter 6, Using the Widget Toolbox.

**WtList Function**

Controls the characteristics of scrolling list widgets.

**Usage**

```
status = WtList(function, widget [, parameters])
```

**Parameters**

*function:*

- `'Add'` — Add specified item(s) to the list.
• ‘Delete’ — Delete specified item(s) from the list.
• ‘DeleteAll’ — Delete all items from the list.
• ‘Deselect’ — Deselect specified item(s) in the list.
• ‘DeselectAll’ — Deselect all items in the list.
• ‘ItemCount’ — Number of items in the list.
• ‘Items’ — Get items in the list.
• ‘Select’ — Select specified item(s) from the list.
• ‘SelectAll’ — Select all items from the list.
• ‘SelectedCount’ — Number of selected items.
• ‘Selected’ — Get selected items.
• ‘Replace’ — Replace specified item(s) in the list.

widget — The list widget ID.

parameters:
• For ‘Add’ — (p1) A string or array of strings; (p2) a long integer representing the position at which to add the item. The default is the end position.
• For ‘Delete’ — (p1) A string or array of strings.
• For ‘Deselect’ — (p1) A string or array of strings.
• For ‘ItemCount’ — (p1) Number of items in the list (long, output).
• For ‘Items’ — (p1) Items in the list. Array of strings (output).
• For ‘Menu’ — (p1) Menu shell ID (long).
• For ‘Select’ — (p1) A string or array of strings.
• For ‘SelectedCount’ — (p1) Number of selected items (long, output).
• For ‘Selected’ — (p1) Selected items. Array of strings (output).
• For ‘Replace’ — (p1) A string or array of strings; (p2) position of first items to be replaced.

Keywords
Notify — If specified, a callback is called during Select or Deselect operations.
Discussion

Windows USERS Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

The list utility lets users choose an action from a list of actions. For detailed information on the list utility, see the description of XmList in the OSF/Motif Programmer’s Reference.

Example

```
items = ['Presidents Day', 'St.Patricks Day', '
        'Easter', 'Memorial Day', '4th of July', 
        'Labor Day', 'Halloween', 'Thanksgiving', 
        'Hanukkah', 'Christmas', 'New Years Eve']
status = WtList("Add", datelist, items)
```

See Also

For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to Chapter 6, Using the Widget Toolbox.
**WtLookupString Function**

Maps a KeyPress or KeyRelease event to its KeyEvent structure (and optionally, to its Keysym) when a user presses a key.

**Usage**

```plaintext
string = WtLookupString(event)
```

**Input Parameters**

`event` — A KeyEvent structure associated with the KeyPress event. The KeyEvent structure is passed to the callback procedure.

**Returned Value**

`string` — A string containing the name of the key which was pressed. A null string is returned for the `<Shift>`, `<Ctrl>`, `<Alt>`, and function keys.

**Keywords**

`Keysym` — Returns a value associated with the key pressed.

---

**UNIX USERS**

For Motif, `Keysym` returns a long value (XKeysym) associated with the key pressed. `Keysym` is an integer value unique to a particular key on the keyboard. This value can be used to identify function key presses.

---

**Windows USERS**

For Windows, `Keysym` returns the long value of the virtual key associated with the key pressed (the result of the VkKeyScan procedure). For a list of virtual key codes and the keys to which they map, see Appendix E. Virtual Keys or refer to the Win32 Programmer’s Reference.

---

**Discussion**

**Windows USERS**

Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.
WtLookupString is used in the callback of a KeyPress or KeyRelease event to map the event into the string it represents. The call is used specifically to handle null strings, which are returned whenever an unprintable character (such as \textless Shift\rangle, or a function key) is pressed.

The \textit{Keysym} keyword is used in situations where the pressed key must be identified. The include file <\texttt{X11/keysymdef.h}> contains a complete Keysym listing.

**Examples**

The following example illustrates the use of WtLookupString to obtain the string name associated with a KeyPress event.

```c
string = WtLookupString(event)
```

The following usage of WtLookupString returns the string associated with the event as well as the Keysym associated with the key pressed.

```c
string = WtLookupString(event, keysym=ks)
```

**See Also**

WtAddHandler

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, \textit{Using WAVE Widgets}.


---

**WtLoop Procedure**

Handles the dispatching of events and calling of callback routines.

**Usage**

WtLoop

**Parameters**

None.
Returned Value

None.

Keywords

*Noblock* — If specified and nonzero, events are dispatched in the background, and WtLoop returns immediately to process PV-WAVE commands from the command line.

Discussion

WtLoop causes PV-WAVE to loop indefinitely, processing the events and dispatching callbacks, handlers, and timers.

*Windows USERS* Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

**NOTE** For interapplication development using `cwavec()` or `cwavefor()`, the default behavior of WwLoop is to block, even if the *Noblock* keyword is set. To force nonblocking, set `Noblock = 2` and then call `WtProcessEvent` periodically to service the event loop.

Motif Example

This example creates a Motif button labeled **Done**.

```
PRO example
  COMMON block, top
  @txmclasses.pro
  top=WtInit('wt_ex4', 'Examples')
  widget=WtCreate('Done', xmPushButtonWidgetClass, top)
  status=WtAddCallback(widget, 'activateCallback', 'CancelHelp')
  status=WtSet(top, /Realize)
  WtLoop
END
```
**WtMainLoop Function**

Handles the dispatching of events.

**Usage**

\[ \text{status} = \text{WtMainLoop}( ) \]

**Input Parameters**

None.

**Output Parameters**

None.

**Returned Value**

\[ \text{status} \] — One (1) indicates success; zero (0) indicates failure.

**Keywords**

*Noblock* — If specified and nonzero, events are dispatched in the background, and WtMainLoop returns immediately to process PV-WAVE commands from the command line.

**Discussion**

*Windows USERS*  
Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

The WtLoop function can be called to accomplish the same result as WtMainLoop. WtMainLoop has been retained to provide upward compatibility with an earlier release of PV-WAVE, but it is recommended that you use WtLoop, instead.
See Also

WtLoop

For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to Chapter 6, Using the Widget Toolbox.

WtPointer Function

The pointer utility function.

Usage

\[
\text{status} = \text{WtPointer}(\text{function}, \text{widget} [, \text{parameters}])
\]

Parameters

function:

- `'GetLocation'` — Get the current location of the pointer.
- `'GetControl'` — (Motif only) Get the control attributes of the pointer.
- `'GetMapping'` — (Motif only) Get the current mapping of the pointer.
- `'SetLocation'` — (Motif only) Set the location of the pointer.
- `'SetControl'` — (Motif only) Set the control attributes of the pointer.
- `'SetMapping'` — (Motif only) Set the mapping of the pointer.

widget — The current widget.

parameters:

- `'GetLocation'` — A seven-element array of long integers:

  0  Root window
  1  Child window where the pointer is located
  2  X coordinate in root
  3  Y coordinate in root
  4  X coordinate in current window
  5  Y coordinate in current window
6 Modifier keys and buttons state mask

• ‘GetControl’ — A three-element array of long integers:

0 accel_numerator
1 accel_denominator
2 threshold

• ‘GetMapping’ — An array of up to ten long integers:

0–9 Pointer button mapping

• ‘SetLocation’ — A two element array of long integers:

0 X coordinate in current window
1 Y coordinate in current window

• ‘SetControl’ — A three-element array of long integers:

0 accel_numerator
1 accel_denominator
2 threshold

• ‘SetMapping’ — An array of up to ten long integers:

0–9 Pointer button mapping

Returned Value

status — One (1) indicates success; zero (0) indicates failure.
Discussion

Windows USERS  Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.


Example

```
status = WtPointer("GetLocation", wid, state)
```

See Also

For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to Chapter 6, Using the Widget Toolbox.

WtPreview Function

Handles utility functions for the preview widget (XvpvPreview).

Usage

```
status = WtPreview(action, widget)
```

Input Parameters

- **action** — A string containing one of the following actions:
  - AutoDefine — Automatically define the data area.
  - ClearArea — Clear the given data area. If this action is specified, you must also provide the following input parameters:
type — The type of object to commit.

count — The number of areas to clear.

areas — The selected areas to clear.

ClearAll — Clear all of the selected areas.

CommitArea — Commit the selection of the given area. If this action is specified, you must also provide the following input parameters:

  type — The type of object to commit.

  count — The number of areas to commit.

  areas — The selected areas to commit.

SelectArea — Select the given area. If this action is specified, you must also provide the following input parameters:

  type — The type of object to select.

  area — The area to select.

widget — The widget ID of the Preview widget (XvnPreview).

Discussion

Windows USERS  Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

XvnPreview is a widget class that allows you to preview column-oriented ASCII data. WtPreview allows the PV-WAVE application programmer to interact with, and modify the preview widget. Specifically, WtPreview facilitates the selection and clearing of areas in a data file being displayed by the preview widget.

NOTE  See WwPreview Procedure on page 315 for more detailed information on the preview widget and an example preview window application.
XvnPreview Widget Documentation

The XvnPreview widget class was developed by Visual Numerics, Inc.

Complete documentation for the XvnPreview widget, including information on the widget’s resources and callbacks, is available in the PostScript file `preview_motif.ps`, which you can print on any PostScript printer. This file is in:

(UNIX) \( \text{WAVE}_{\text{DIR}}/\text{docs/} \text{widgets} \)

(OpenVMS) \( \text{WAVE}_{\text{DIR}}: [\text{DOCS.} \text{WIDGETS}] \)

(Windows) \( \text{WAVE}_{\text{DIR}}/\text{docs/} \text{widgets} \)

Where \( \text{WAVE}_{\text{DIR}} \) is the main PV-WAVE directory.

See also Appendix B, Motif Callback Parameters for information on the required parameters for all widget callbacks.

Example

See the example for WwPreview.

See Also

WwPreview, WzPreview (in the PV-WAVE Reference)

For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to Chapter 6, Using the Widget Toolbox.

WtProcessEvent Function

Handles the dispatching of a Widget Toolbox event.

Usage

\[
\text{status} = \text{WtProcessEvent}( )
\]

Input Parameters

None.
Returned Value

`status` — A value indicating the success or failure of the processed event as follows:

- 1 Indicates the event loop is stopped.
- 0 Indicates the event was processed.
- −1 Indicates the keyboard was received.

Keywords

`Drain` — Causes all pending events to be flushed.

Discussion

Windows USERS Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

WtProcessEvent processes one Widget Toolbox event. Use this function to process Widget Toolbox events in the user-customized event loop, or to temporarily halt the execution of the PV-WAVE program while still processing Widget Toolbox events.

WtProcessEvent normally processes a single event. If there are no pending events, then it will wait (block) for the next event. If you use the `Drain` keyword, then WtProcessEvent will process all pending events and then return. If no events are pending, it returns immediately, without blocking.

Example

WtProcessEvent is used in the source code for the WwAlert function to process events until the user presses a button, or until the event processing fails. The source code for WwAlert is in:

(UNIX)     WAVE_DIR/lib/std/motif/wwalert.pro
(OpenVMS)  WAVE_DIR:[LIB.STD.MOTIF]WWALERT.PRO
(Windows)  WAVE_DIR\lib\std\windows\wwalert.pro
Where \texttt{WAVE\_DIR} is the main PV-WAVE directory.

\section*{See Also}
For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, \textit{Using WAVE Widgets}.

\section*{WtResource Function}
Queries, creates, saves, or modifies the widget resource database.

\section*{Usage}
\begin{verbatim}
value = WtResource([resvar])
\end{verbatim}

\section*{Input Parameters}
\begin{itemize}
\item \texttt{resvar} --- (optional) A string containing a resource variable specification in the resource database.
\end{itemize}

\section*{Returned Value}
\begin{itemize}
\item \texttt{value} --- The returned value depends on the input parameter and/or the use of keywords as shown in the following table.
\end{itemize}

\begin{tabular}{|l|l|}
\hline
\textbf{Value Returned} & \textbf{Parameter or Keyword Used} \\
\hline
A string containing the value associated with \texttt{resvar}; a null string, if \texttt{resvar} isn't found; or the default value specified. & \texttt{resvar}, or \texttt{resvar} with \texttt{Default} keyword \\
An integer value of 1 indicating success, or 0 indicating failure. & \texttt{Add}, \texttt{Load}, or \texttt{Save} keywords \\
A string containing the resource specification a widget. & \texttt{Spec} keyword \\
\hline
\end{tabular}
Keywords

Add — A string containing a name: value resource specification to merge into the resource database in the current session. If the resource specification already exists in the application, the Add keyword takes precedence.

Default — (Used only if resvar is specified.) A string specifying a default value for the resource variable in resvar. If resvar doesn’t match anything in the resource database, this default value is returned.

Load — A string specifying the pathname of a resource file to be merged with the existing resource database. If the resource file was specified in the application, the use of the Load keyword takes precedence.

Save — A string specifying the pathname of the resource file in which to save the currently defined resources in the resource database. If the file specified already exists, the contents will be overwritten.

Spec — Used to specify a widget ID. The resource variable specification of the widget is returned.

Discussion

Windows USERS  Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

WtResource gives direct access to the widget resource database from WAVE Widgets and Widget Toolbox applications. Resources loaded or added using WtResource are used from then on by all subsequently created WAVE Widgets. The resource specifications merged into the resource database using either the Load or Add keywords supercede existing definitions.

Examples

The following example shows how a resource specification added during a session takes precedence over the existing definition.

```plaintext
myapp*background: red
   ; The existing resource specification in the database.
value = WtResource(ADD='myapp*background: blue')
```
The `Add` keyword changes the color to blue instead. For the rest of the session, or until it is redefined, the background color will be blue.

In the next example, the syntax for using the `Load` keyword is illustrated.

```plaintext
value = WtResource(LOAD='~/usr/mydir/myapp/myapp.ad')
; This merges the resources found in the specified file with the existing resource database.
```

This example shows how to request a value associated with a resource name.

```plaintext
value = WtResource('myapp*mybutton.label', Default='my label')
; This usage returns the value associated with the resource variable specified. The `Default` keyword is used to return the value 'my label', if the named resource doesn’t exist.
```

An example for writing all defined resources to a file is:

```plaintext
value = WtResource(SAVE='~/usr/mydir/myapp/myapp.newad')
```

To request the resource variable specification of a particular widget identified by its associated widget ID, use:

```plaintext
value = WtResource(SPEC=widget_id)
```

See Also

`WwResource`

For more information about how to write an application program based on WAVE Widgets, refer to Chapter 5, *Using WAVE Widgets*.

---

### WtSet Function

Sets widget resources.

#### Usage

```plaintext
status = WtSet(widget, [argv])
```

#### Parameters

- `widget` — The widget ID.
- `argv` — (optional) An unnamed structure specifying the resources for the widget or shell.
Returned Value

*status* — One (1) indicates success; zero (0) indicates failure.

Keywords

Seed — Appends a command to the command widget.

*Callback=reason* — Calls all defined callbacks for the specified reason for this widget.

Destroy — Destroys the widget and all children of the widget.

*Error=errmsg* — Displays an error message in the history area of the command widget.

Manage — Manages the given widget (WtCreate manages widgets by default).

Map — Maps the given widget.

MultiClick — Sets the time interval (in milliseconds) that is used to determine if consecutive mouse button clicks are to be interpreted as multiple clicks.

Nonsensitive — Sets the given widget to nonsensitive.

Popup=Event — Pops up the given menu (event specifies the location).

Raise — If present and nonzero, pops the given widget in front of other windows displayed on the monitor.

Realize — Realizes (displays) the given widget.

Realize=grab — Realizes the given shell with one of these grab values:

Xtgrabnone

Xtgrabnonexclusive

Xtgrabexclusive

Remove_callback = name — If the name of a callback routine is specified, this keyword removes the named callback. If name is not specified, all callbacks for the specified widget are removed.

Remove_handler = name — If the name of an event handler is specified, this keyword removes the named event handler. If name is not specified, all event handlers for the specified widget are removed.

Search=dir — (Motif only) Sets the search context for a file selection widget.

Sensitive — Sets the given widget to sensitive.
**Unmanage** — Unmanages the given widget.

**Unmap** — Unmaps the given widget.

**Unrealize** — If present and nonzero, unrealizes a widget, or pops down (undisplays) a shell.

**Update** — If present and nonzero, all pending exposure (i.e., window repair) events are processed immediately. If you suspect that a callback procedure will take a long time, use this keyword to update the display before starting the time consuming operation.

**Userdata** — Stores a copy of the value of a variable.

**Value=value** — Sets the value for a command, scale, scroll bar, or toggle button.

**Whenmapped** — Sets mapped to true for managed attributes.

**Whenunmapped** — Sets mapped to false for managed attributes.

**Discussion**

**Windows USERS** Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

**Windows USERS** The WtSet keywords are supported for Windows; however, the use of the *argv* parameter to pass resource-value pairs is not supported.

Resources are passed as an unnamed structure, where tag names correspond to resource names and tag definitions are resource values. Resource names are given without the XmN (Motif) prefix.

An unnamed structure has the following general definition:

\[
x = \{, \text{tag}_1 \text{=} \text{tag}_1, \text{tag}_2 \text{=} \text{tag}_2, \ldots, \text{tag}_n \text{=} \text{tag}_n, \}
\]

See the *PV-WAVE Programmer’s Guide* for detailed information on unnamed structures.

For a list of resources for the specified widget or shell, see the appropriate *OSF/Motif Programmer’s Reference*. 
NOTE  If the color, font, or bitmap (called a “pixmap” in Motif) resource value is of type string, it is assumed to be the value of the color/font name or bitmap filename, and the appropriate resources are loaded.

Example

```c
args={,x:100,y:100,label:'Enter File:',$ string:'/usr/home/myfile.pro'}
; Resources are defined in an unnamed structure.
status=WtSet(w,args)
```

See Also

For more information about how to write an application program based on the PIV-WAVE Widget Toolbox, refer to Chapter 6, Using the Widget Toolbox.

### WtTable Function

Modifies an XbaeMatrix class widget.

#### Usage

```c
status = WtTable(function, widget [, parameters])
```

#### Input Parameters

- **function:**
  
  ‘AddColumns’ — Add columns to the table. If this function is specified, also provide the following parameters:
**position** — The column before which new columns are added.

**columns** — Two-dimensional string array of column values.

**num_columns** — Number of columns to be added.

**width** — One-dimensional array of column widths.

**labels** — (optional) One-dimensional array of column labels.

**max_lengths** — (optional) One-dimensional array of column maximum lengths.

**alignments** — (optional) One-dimensional array of column alignments. Valid values are:

0       Align cell contents to cell’s left edge (left justify).
1       Center cell contents (center justify).
2       Align cell contents to cell’s right edge (right justify).

**label_alignments** — (optional) One-dimensional array of column label alignments. Valid values are:

0       Align cell contents to cell’s left edge (left justify).
1       Center cell contents (center justify).
2       Align cell contents to cell’s right edge (right justify).

**colors** — (optional) Two-dimensional array of column cell colors.

`AddRows` — Add rows to the table. If this function is specified, also provide the following parameters:

**position** — The row before which new rows are added.

**rows** — Two-dimensional string array of row values.

**num_rows** — Number of rows to be added.

**labels** — (optional) One-dimensional array of row labels.

**colors** — (optional) Two-dimensional array of row cell colors.
‘DeleteCols’ — Delete columns from the table. If this function is specified, also provide the following parameter:

\[\text{position}\] — The column at which to start deleting columns.

\[\text{num\_columns}\] — Number of columns to delete.

‘DeleteRows’ — Delete rows from the table. If this function is specified, also provide the following parameter:

\[\text{position}\] — The row at which to start deleting rows.

\[\text{num\_rows}\] — Number of rows to delete.

‘DeselectAll’ — Deselect all cells in the table.

‘DeselectCell’ — Deselect the specified cell in the table. If this function is specified, also provide the following parameter:

\[\text{row}\] — Row index of the cell to deselect.

\[\text{column}\] — Column index of the cell to deselect.

‘DeselectCol’ — Deselect the specified column in the table. If this function is specified, also provide the following parameter:

\[\text{column}\] — Index of column to deselect.

‘DeselectRow’ — Deselect the specified row in the table. If this function is specified, also provide the following parameter:

\[\text{row}\] — Index of row to deselect.

‘SelectCell’ — Select the specified cell in the table. If this function is specified, also provide the following parameters:

\[\text{row}\] — Row of the cell to select.

\[\text{column}\] — Column of the cell to select.

‘SelectCol’ — Select the specified column in the table. If this function is specified, also provide the following parameter:

\[\text{column}\] — Column to select.

‘SelectRow’ — Select the specified row in the table. If this function is specified, also provide the following parameter:

\[\text{row}\] — Row to deselect.
'EditCell' — Edit the specified cell in the table. If this function is specified, also provide the following parameters:

row — Row of the cell to edit.

column — Column of the cell to edit.

'CancelEdit' — Cancel the edit of the currently edited cell. If this function is specified, also provide the following parameter:

unmap — (optional) If specified and nonzero, the currently edited cell is unmapped.

'CommitEdit' — Commit the edit of the currently edited cell. If this function is specified, also provide the following parameter:

unmap — (optional) If specified and nonzero, the currently edited cell is unmapped.

'GetCell' — Get the value of a cell in the table. If this function is specified, also provide the following parameters:

row — Row of the cell to get.

column — Column of the cell to get. Returns the value of the specified cell.

'SetCell' — Set the value of a cell in the table. If this function is specified, also provide the following parameters:

row — Row of the cell to set.

column — Column of the cell to set.

t value — New value of the specified cell.

'SetColor' — Set the color value for a cell in the table. If this function is specified, also provide the following parameters:

row — Row of the cell whose color you want to set.

column — Column of the cell whose color you want to set.

color — New colormap index of the specified cell.

'SetColColor' — Set the color value for a column in the table. If this function is specified, also provide the following parameters:

position — Column at which to start setting colors.

colors — Two-dimensional array of new colormap indexes for columns.
num_colors — The number of elements in the colors array.

'SetRowColor' — Set the color value for a row in the table. If the
'SetRowColor' function is specified, also provide the following parameters:

position — Row at which to start setting colors.

colors — 2D array of new colortable indexes for rows.

num_colors — The number of colors in the colors array.

widget — Widget ID of the table (xbaeMatrix class) widget.

Discussion

Windows USERS Visual Numerics has ported a subset of the Widget Toolbox
(Wt) functionality that is available for Motif to Microsoft Windows. Because the
Widget Toolbox under Windows is not a complete implementation, we recommend
that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools
when developing GUI applications.

The XbaeMatrix widget creates an editable 2D array of string data (cells) similar
to a spreadsheet. All values must be converted to string type before being set with
WtTable.

XbaeMatrix Widget Documentation

The Motif version of the XbaeMatrix widget was originally developed by Andrew
Wason of Bellcore.

Complete documentation for the XbaeMatrix widget is available in the PostScript
file matrix_motif.ps, which you can print on any PostScript printer. This file
is in:

(UNIX)  WAVE_DIR/docs/widgets

(OpenVMS)  WAVE_DIR:[DOCS.WIDGETS]

(Windows)  WAVE_DIR\docs\widgets

Where WAVE_DIR is the main PV-WAVE directory.

Refer to this document for detailed information on the XbaeMatrix widget’s
resources and callbacks.
**XbaeMatrix Widget Callbacks**

See Appendix B, *Motif Callback Parameters* for information on the required parameters for all widget callbacks. The XbaeMatrix widget’s callback routines and their parameters are documented in the file `matrix_motif.ps` and discussed in the previous section. Refer to that document for information on the table widget’s callbacks.

**Example**

Values in the table are modified (set) through a call to WtTable.

![Figure 8-1 Editable table based on the XbaeMatrix widget.](image)

```pro
PRO ButtonCB, w, which
  COMMON Widgets, table
  ; When a button is pressed, modify the table with a call to WtTable.
  CASE which OF
    1: PRINT, WtGet(table, 'cells', Nrows = 2, Ncols = 3)
    2: status = WtTable('SetCell', table, 0, 0, 'New Font')
  ENDCASE
END

PRO LeaveCellCB, w, data, n, reason, event,$
  r, c, value, doit
  ; This callback is called just before an edit to a cell is committed. For more information on XbaeMatrix widget
  ; callbacks and their parameters, see the previous section “XbaeMatrix Widget Callbacks”.
  PRINT, "leaveCellCallback ", r, c, value
END```

406 Application Developer’s Guide
PRO ModifyVerifyCB, w, data, n, reason, event,$
   r, c, doit, ci, cn, sp, ep, txt, len, fmt
   ; This callback is called while a cell is being edited. For more
   ; information on XbaeMatrix widget callbacks and their
   ; parameters, see the previous section “XbaeMatrix
   ; Widget Callbacks”.
   print, "modifyVerifyCallback ", r, c, doit,$
      ci, cn, sp, ep, txt, len, fmt
END

PRO Table
   @wtxmclasses
   @wtxmconsts
   COMMON Widgets, table
   ; Initialize the Widgets Toolkit.
   top = WwInit('table','Test', layout, /Vertical)
   ; Load color table.
   loadct, 5
   ; Set cell contents, display table.
   cells = [['Fonts','Size','Icons'],
      ['1.245','2.5','3.6']]
   args = {, rows:2, columns:3, cells:cells,$
      columnWidths:[10, 10, 10],$
      columnLabels:['col1','col2','col3'],$
      rowLabels:['row1','row2'], boldLabels:TRUE,$
      colors:[[40, 50, 60], [100, 110, 120]],$
      columnAlignments:[XmALIGNMENT_BEGINNING, $
         XmALIGNMENT_CENTER, XmALIGNMENT_END]$
   table = WtCreate('table', xbaeMatrixWidgetClass,  $
      layout, args)
   status = WtAddCallback(table, $
      "leaveCellCallback", 'leaveCellCB')
   status = WtAddCallback(table, "modifyVerifyCallback",$
      'modifyVerifyCB')
   buttons = WwButtonBox(lay out, ['Get','Set'], 'ButtonCB')
   status = WwSetValue(top, /Display)
   WwLoop
END

See Also

WwTable
For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to Chapter 6, Using the Widget Toolbox.

---

**WtTimer Function**

Registers a callback function for a given timer.

**Usage**

\[
\text{status} = \text{WtTimer}(\text{function, params, [client\_data]})
\]

**Parameters**

- **function**: 
  - 'Add' — If Add is specified, also provide the following three parameters:
    - time — A long integer specifying the time interval in milliseconds.
    - timer — A string containing a timer function name.
    - client\_data — (optional) A variable.
  - 'Remove' — If Remove is specified, also provide the following parameter:
    - id — The timer ID.

**Returned Value**

For 'Add':

- status — The input source ID, or zero (0) to indicate failure.

For 'Remove':

- status — One (1) indicates success; zero (0) indicates failure.

**Keywords**

- **Noparams** — If nonzero, calls the timer with two parameters: wid (the top-level widget name), and client\_data.
- **Once\_only** — If present and nonzero, the timer function is called only once.
Discussion

Windows USERS  Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

This timer, unlike the XtIntrinsics XtTimeOut function, restarts itself. To stop the timer, use the command:

\[
\text{WtTimer("REMOVE", id)}
\]

in the timer callback. The new timer ID is returned in the timer callback parameter `timer_id`.

If you need to keep a copy of the timer ID in a common block, copy the `timer_id` parameter of the timer callback routine to your timer ID variable in the COMMON block, as shown in the following example.

For information on the requirements for writing timer procedures, see Using the Widget Toolbox on page 217.

Example

COMMON timer, tid

id=WtTimer("ADD", 100, 'TimerCallback', my_data)

PRO TimerCallback, wid, client_data, nparams, timer_id, interval
    COMMON timer, tid
    tid = timer_id

END

See Also

For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to Chapter 6, Using the Widget Toolbox.
WtWorkProc Function

Registers a work procedure for background processing.

Usage

\[ \text{status} = \text{WtWorkProc}(\text{function, parameters}) \]

Parameters

function:
- ‘Add’ — Registers the specified work procedure.
- ‘Remove’ — Removes the previously registered work procedure.

parameters:
- If ‘Add’ is specified, also provide these two parameters:
  \[ \begin{align*}
  \text{workproc} & \quad \text{A work procedure name (string).} \\
  \text{client_data} & \quad \text{(optional) A variable.}
  \end{align*} \]
- If ‘Remove’ is specified, also provide the following parameter:
  \[ \text{id} \quad \text{The ID of the work procedure to be removed.} \]

Returned Value

For ‘Add’:
\[ \text{status} \quad \text{The work procedure ID; zero (0) indicates failure.} \]

For ‘Remove’:
\[ \text{status} \quad \text{One (1) indicates success; zero (0) indicates failure.} \]

Keywords

\textbf{Noparams} — If nonzero, calls the work procedure with two parameters: \textit{wid} (the top-level shell of the application), and \textit{client_data}.

\textbf{Once_only} — If nonzero, calls the work procedure one time only.
Discussion

Windows USERS Visual Numerics has ported a subset of the Widget Toolbox (Wt) functionality that is available for Motif to Microsoft Windows. Because the Widget Toolbox under Windows is not a complete implementation, we recommend that Windows developers use the WAVE Widgets (Ww) layer or the VDA Tools when developing GUI applications.

WtWorkProc is modeled after functionality available in the X Windows Intrinsics library with XtAddWorkProc and XtRemoveWorkProc. When a work procedure is added, it is executed in its entirety unless a REMOVE call to WtWorkProc is issued before the procedure has been called. If the work procedure does a large amount of processing it could block the widget interface until it has finished running.

A typical implementation using WtWorkProc to perform a large amount of background processing is to break down the processing into a number of discrete steps and execute the steps one at a time in a work procedure which issues another ADD call to WtWorkProc to start the next step. Repeat this process until all the steps have been completed.

For information on the requirements for writing work procedures, see Adding Work Procedures on page 226.

Example

PRO ButtonCB
    ; This callback is called when a button is selected. Do not initiate the selected;
    ; operation here, because it is too time-consuming. Instead, schedule a work procedure.
    id=WtWorkProc("ADD", ‘MyWorkProc’, my_data)
    .
    .
END

PRO MyWorkProc, wid, client_data, workproc_id
    .
    .
    IF done THEN status = WtWorkProc("REMOVE", workproc_id)
END

See Also

For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to Chapter 6, Using the Widget Toolbox.
This chapter describes the new VDA Tools Manager API routines. These functions allow VDA Tools to communicate with the central VDA Tools core.

**TmAddSelectedVars Procedure**

Adds selected variables from a VDA Tool to the list of selected variables in the Tools Manager.

**Usage**

TmAddSelectedVars, *tool_name*, *var_name*

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool.

*var_name* — A string containing the name of a variable from the VDA Tool to add to the selected variables list.

**Keywords**

None.
**Discussion**

The Tools Manager maintains a list of variables that have been selected in a VDA Tool. The selected variables list enables variables to be exchanged between VDA Tools. This routine is called by the Export Variables dialog box and the data selection function found in VDA Tools.

**Example**

This example demonstrates how variables from a VDA Tool are added to the variable selection list, exported to another VDA Tool, and deleted from the variable selection list.

```plaintext
PRINT, TmEnumerateSelectedVars()
; No variables are currently on the variable selection list.
PRINT, TmEnumerateVars('WzPlot_0')
   HAN
; The variable HAN is currently displayed in the VDA Tool WzPlot_0.
TmAddSelectedVars, 'WzPlot_0', 'HAN'
; Add the variable HAN in the VDA Tool WzPlot to the selected variables list.
PRINT, TmEnumerateSelectedVars()
   HAN
; Enumerate the variables that are on the selected variables list.
TmExportSelection, ['WzPlot_2']
; Export the variable on the selection list to another active VDA Tool.
TmDeselectVars
; Remove all variables from the variable selection list.
```

**See Also**

TmDeselectVars, TmEnumerateSelectedVars, TmExport, TmExportSelection
**TmAddVar Procedure**

Adds a variable to a VDA Tool.

**Usage**

TmAddVar, *tool_name*, *var_name*

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool.

*var_name* — The name of a variable to add to the VDA Tool.

**Keywords**

None.

**Discussion**

This function registers a variable with the Tools Manager and associates the variable with a specific VDA Tool instance. VDA Tools can use the UPVAR, INFO, and TmGetVarMainName functions to get the names of variables at the $MAIN$ program level.

**Example**

The following calls can be used to retrieve the name of a variable that was passed to the VDA Tool and then add that variable to the Tools Manager.

```
INFO, var
A      INT    = Array(100)
B      FLOAT  = Array(30)
TmAddVar, ‘WzPlot_0’, ‘A’
PRINT, TmEnumerateVars(‘WzPlot_0’)
```

**See Also**

INFO (in the *PV-WAVE Reference*), UPVAR (in the *PV-WAVE Reference*), TmDelVar, TmEnumerateVars, TmGetVarMainName
**TmCodeGen Procedure**

Writes a specified string to the code generation file.

**Usage**

TmCodeGen, *string*

**Parameters**

*string* — A string to write into the code generation file.

**Keywords**

None.

**Discussion**

The code generation functions allow a VDA Tool user to write the PV-WAVE code used to create a plot, import data, or any other VDA Tool action.

This procedure is used in the TM_CODEGEN method to write strings containing PV-WAVE code to a file. The TmStartCodeGen command opens the file, and the TmEndCodeGen closes the file when writing is completed. The TM_CODEGEN method is called by default from the **File=>Code Generation** menu item.

**Example**

For an example of how the code generation functions are used, look at the source file for the WzPlot VDA Tool. In that file, study the procedure WzPlotCodeGen to learn how TmCodeGen is used to write PV-WAVE code to a file. The source file for WzPlot is in:

- (UNIX) `<wavedir>/lib/vdatools/wzplot.pro`
- (OpenVMS) `<wavedir>:[LIB.VDATOOLS]WZPLOT.PRO`
- (Windows) `<wavedir>\lib\vdatools\wzplot.pro`

Where `<wavedir>` is the main PV-WAVE directory.

**See Also**

TmEndCodeGen, TmStartCodeGen
TmCopy Procedure

Copies the selected graphical elements from the specified VDA Tool to the clipboard.

Usage

TmCopy, tool_name

Parameters

(tool_name) — A string containing the unique name of a VDA Tool containing the graphical element that is being copied.

Keywords

None.

Discussion

The TmAddSelectedGrael function is used to add graphical elements to a selection list maintained by the Tools Manager. Once on the selection list, graphical elements can be copied, pasted, cut, or deleted. TmCopy copies the graphical element or elements on the current selection list to the clipboard (a temporary buffer).

The (tool_name) parameter for TmCopy must be the same as the tool name specified in the TmDelSelectedGraels function.

TmCopy is called by the Edit=>Copy function and by the Copy button on the Button Bar.

Example

The following commands obtain the names of the graphical elements registered for a VDA Tool, add one of the graphical elements to the selection list, then copy and paste the graphical element (a rectangle).

PRINT, TmEnumerateGraels(‘WzPlot_0’)  
TM_WINDOWID TM RECTANGLE LINE AXIS LEGEND TEXT MENUBAR  
   BUTTONBAR MESSAGE XX TM_HELP AXIS_0 AXIS_1 TM_DRAWING  
   RECTANGLE_0 LINE_1 GROUP_0  
TmAddSelectedGrael, ‘WzPlot_0’, ‘RECTANGLE_0’
PRINT, TmEnumerateSelectedGraels(‘WzPlot_0’)  
RECTANGLE_0  
TmCopy, ‘WzPlot_0’  
TmPaste, ‘WzPlot_0’

**See Also**

TmCut, TmDelete, TmPaste

---

**TmCut Procedure**

Cuts the selected graphical elements from the specified VDA Tool and moves them to the clipboard.

**Usage**

TmCut, *tool_name*

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool containing the graphical element that is being cut.

**Keywords**

None.

**Discussion**

The TmAddSelectedGrael function is used to add graphical elements to a selection list maintained by the Tools Manager. Once on the selection list, graphical elements can be copied, pasted, cut, or deleted. TmCut cuts the graphical element or elements on the current selection list and moves them to the clipboard (a temporary buffer).

The *tool_name* parameter for TmCut must be the same as the tool name specified in the TmAddSelectedGrael function.

TmCut is called by the **Edit=>Cut** function and by the **Cut** button on the Button Bar.
Example

The following commands obtain the names of the graphical elements registered for a VDA Tool, add one of the graphical elements to the selection list, then cut graphical element (a rectangle).

```plaintext
PRINT, TmEnumerateGraels('WzPlot_0')
TM_WINDOWID TM RECTANGLE LINE AXIS LEGEND TEXT MENUBAR
    BUTTONBAR MESSAGE XX TM_HELP AXIS_0 AXIS_1 TM_DRAWING
    RECTANGLE_0 LINE_1 GROUP_0
TmAddSelectedGrael, 'WzPlot_0', 'RECTANGLE_0'
PRINT, TmEnumerateSelectedGraels('WzPlot_0')
    RECTANGLE_0
TmCut, 'WzPlot_0'
```

See Also

TmCopy, TmDelete, TmPaste

---

**TmDelVar Procedure**

Removes variables from a VDA Tool.

**Usage**

```plaintext
TmDelVar, tool_name [, var_names]
```

**Parameters**

- `tool_name` — A string containing the unique name of a VDA Tool.
- `var_names` — (optional) A string array containing the names of the variables to remove from the variable list of the specified VDA Tool.

**Keywords**

- `All` — If nonzero, removes all the variables on the variable list of the VDA Tool. This keyword takes precedence over the `var_names` parameter.
Discussion
If neither the var_names parameter nor the All keyword is specified, no action is taken.

Example
var_names = ['var1', 'var2', 'var3']
TmAddVar, unique, var_names
TmDelVar, unique, var_names

See Also
TmAddVar, TmEnumerateVars, TmGetVarMainName

TmDelete Procedure
Permanently deletes the selected graphical elements from the specified VDA Tool.

Usage
TmDelete, tool_name

Parameters
tool_name — A string containing the unique name of a VDA Tool from which the graphical elements will be deleted.

Keywords
None.

Discussion
The TmAddSelectedGrael function is used to add graphical elements to a selection list maintained by the Tools Manager. Once on the selection list, graphical elements can be copied, pasted, cut, or deleted. TmDelete cuts the graphical element or elements on the current selection list. The deleted elements are not moved to the clipboard; thus, they cannot be pasted back into a VDA Tool window.
The tool_name parameter for TmDelete must be the same as the tool name specified in the TmAddSelectedGrael function.

TmDelete is called by the Edit=>Delete function and by the Delete button on the Button Bar.

**Example**

The following commands obtain the names of the graphical elements registered for a VDA Tool, add one of the graphical elements to the selection list, then delete the graphical element (a rectangle).

```plaintext
PRINT, TmEnumerateGraels('WzPlot_0')
TM_WINDOWID TM RECTANGLE LINE AXIS LEGEND TEXT MENUBAR
   BUTTONBAR MESSAGE XX TM_HELP AXIS_0 AXIS_1 TM_DRAWING
   RECTANGLE_0 LINE_1 GROUP_0
TmAddSelectedGrael, 'WzPlot_0', 'RECTANGLE_0'
PRINT, TmEnumerateSelectedGraels('WzPlot_0')
   RECTANGLE_0
TmDelete('WzPlot_0')
```

**See Also**

TmCopy, TmCut, TmPaste

---

**TmDeselectVars Procedure**

Clears the current list of selected variables.

**Usage**

TmDeselectVars

**Parameters**

None.

**Keywords**

None.
Discussion

This procedure is used to remove variables from the list of selected variables. Usually, this routine is called before a new variable selection is made with TmAddSelectedVars.

Example

This example demonstrates how variables from a VDA Tool are added to the variable selection list, exported to another VDA Tool, and deleted from the variable selection list.

```
PRINT, TmEnumerateSelectedVars()
; No variables are currently on the variable selection list.
PRINT, TmEnumerateVars('WzPlot_0')
   HAN
; The variable HAN is currently displayed in the VDA Tool WzPlot_0.
TmAddSelectedVars, 'WzPlot_0', 'HAN'
; Add the variable HAN in the VDA Tool WzPlot to the selected variables list.
PRINT, TmEnumerateSelectedVars()
   HAN
; Enumerate the variables that are on the selected variables list.
TmExportSelection, ['WzPlot_2']
; Export the variable on the selection list to another active VDA Tool.
TmDeselectVars
; Remove all variables from the variable selection list.
PRINT, TmEnumerateSelectedVars()
```

See Also

TmAddSelectedVars, TmEnumerateSelectedVars, TmExport, TmExportSelection
TmDynamicDisplay Procedure

Displays selected data in all active VDA Tools, provided that the VDA Tools can display the related variable(s).

Usage

TmDynamicDisplay, indices

Parameters

indices — An associative array containing the names of one or more selected variables and the indices that were selected for each variable. This associative array is built by the TM_SELECTED_DATA method procedure. The variable names are the associative array keys and the indices are the associative array’s values.

Keywords

None.

Discussion

If you want the data that is selected in one VDA Tool to be automatically selected (and highlighted) in other VDA Tools, use the routine TmDynamicDisplay.

TmDynamicDisplay is used in conjunction with the TM_DATA_SELECTION method. Whenever data is selected in a VDA Tool, the TM_DATA_SELECTION method procedure is called. You must decide what action this method procedure takes. For instance, if you want the data selection to appear in all active VDA Tools that can display the selected variable, then the method procedure must do the following:

• Build an associative array containing the names of all the selected variables and the selected indices for each variable.
• Call TmDynamicDisplay.

TmDynamicDisplay then does the following:

• Uses the keys and values of the associative array to set the GLOBAL.VARIABLE.INDICES attribute for each selected variable.
• Uses the keys of the associative array to identify which active VDA Tools contain the selected variables.
• Executes the TM_DISPLAY method procedure for each of these identified VDA Tools where TM_DISPLAY is set. (The TM_DISPLAY method procedure must query the GLOBAL.VARIABLE.INDICES attribute to get the indices to display for each variable.)

For instance, you can use the TmGetAttributes function to return the indices (a 200-element array):

```
INFO, TmGetAttribute('GLOBAL', 'X', 'INDICES')
Array(199)
```

If the TM_SELECTED_DATA method procedure calls TmDynamicDisplay, the method procedure must distinguish among four possible cases:

• **No Selection** — If no data was selected, then the TM_SELECTED_DATA method procedure returns an empty associative array. When TmDynamicDisplay is called, nothing happens.

• **Single Point Selection** — If a single point is selected in the VDA Tool, then the array built by the TM_SELECTED_DATA method procedure contains the name of one variable (the key) and a single integer (the value). When TmDynamicDisplay is called, the selected point is highlighted in every active VDA Tool that is associated with the selected variable.

• **Multiple Point Selection** — If more than one data point is selected, the array built by the TM_SELECTED_DATA method procedure contains the name of each selected variable and the selected indices from each selected variable. When TmDynamicDisplay is called, the selected data is highlighted in every active VDA Tool that is associated with the selected variable.

• **Area Selection** — In some VDA Tools, such as image display tools, it makes more sense to pass the coordinates of the selection rectangle, rather than all of the selected points, to TmDynamicDisplay. In this case, the associative array contains the selected variable(s) and corresponding 2x2 arrays containing the upper-left and lower-right vertices of the selection rectangle. When TmDynamicDisplay is called, the selected region is highlighted in every active VDA Tool that is associated with the selected variable.

**Example**

```lisp
indices = ASARR(var_name, index)
TmDynamicDisplay, indices
```

**See Also**

ASARR (in the *PV-WAVE Reference*)
**TmDynamicShowVars Procedure**

Updates the variable list.

**Usage**

TmDynamicShowVars

**Parameters**

None.

**Keywords**

*except* — A string specifying the name of a VDA Tool that you do not which to update.

**Discussion**

This procedure dynamically refreshes new variables and selected variables for a VDA Tool. The VDA Tool WzVariable, in particular, uses this procedure.

---

**TmEndCodeGen Procedure**

Closes the file in which generated code is written.

**Usage**

TmEndCodeGen

**Parameters**

None.

**Keywords**

None.
Discussion
The code generation functions allow a VDA Tool user to write the PV-WAVE code used to create a plot, import data, or any other VDA Tool action.

TmEndCodeGen closes the code generation file and frees its logical unit number (LUN). The code generation file is opened with TmStartCodeGen, and strings are written to the file with TmCodeGen. When the writing is completed, call TmEndCodeGen to write an end statement to the file and close it.

This code generation routine must be used after execution of the method TM_CODEGEN.

TmEndCodeGen called from the File=>Code Generation menu item.

Example
TmEndCodeGen

See Also
TmCodeGen, TmStartCodeGen

TmEnumerateAttributes Function
Obtains the attributes for a specified graphical element, variable, or other item in a VDA Tool.

Usage
attr_list = TmEnumerateAttributes(tool_name, item)

Parameters
tool_name — A string containing the unique name of a VDA Tool.

item — A string containing the name of a graphical element, variable, or other item in the VDA Tool.

Returned Value
attr_list — A string array containing the names of the attributes that were set for the specified item.
Keywords
None.

Discussion
Items like graphical elements and variables can have attributes associated with them. Each attribute can have a value. For example, a line’s attributes might include linestyle, line thickness, color, and others. This function simply lists all the attributes associated with a specified item in a VDA Tool.

To obtain a list of all the items currently associated with a VDA Tool, use TmEnumerateGraels. To obtain a list of variables, use TmEnumerateVars.

Example
The following commands list the graphical elements and items associated with a VDA Tool. Then, the attributes for one of those items, an axis, is printed using TmEnumerateAttributes. Each attribute has a value that can be obtained with TmGetAttribute and set with TmSetAttribute.

PRINT, TmEnumerateGraels('WzPlot_2')
   TM_WINDOWID TM_RECTANGLE LINE AXIS LEGEND TEXT MENUBAR
   BUTTONBAR MESSAGE WZPLOT_2_VAR0 TM_HELP AXIS_0 AXIS_1 TM_DRAWING
   ; List the graphical elements and items associated with the unique VDA Tool
   ; WzPlot_2.
PRINT, TmEnumerateAttributes('WzPlot_2', 'AXIS_0')
   BMARGIN STYLE TM_TYPE TMARGIN SIZE ISLIT MOTION NB_HANDLES FONT
   NMIN THICK LMAJ MIN PLACE COORD_SYS LSTYLE RANGE NMAJ MAX COORD
   COLOR
   ; List the attributes associated with the graphical element AXIS_0 that is
   ; associated with WzPlot_2.

See Also
TmGetAttribute, TmSetAttribute
**TmEnumerateItems Function**

Obtains the items defined for a specified VDA Tool.

**Usage**

\[
item\_list = \text{TmEnumerateItems}(\text{tool\_name})
\]

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool.

**Returned Value**

*item_list* — A string array containing the names of the items that were defined for the given VDA Tool.

**Keywords**

None.

**Discussion**

Items can include a wide variety of things that you want to put in a VDA Tool. Some examples of items are:

- variables
- help files
- drawing area
- file names

This function simply lists all the items associated with a specified VDA Tool.

**Example**

The following command lists the items associated with a VDA Tool.

\[
\text{WzHistogram}, \text{HANNING}(30, 30)
\]

\[
\text{PRINT}, \text{TmEnumerateItems}('\text{WzHistogram}_0')
\]

\[
\text{LINE} \text{STATUS} \text{AXIS}_0 \text{T_M\_DRAWING} \text{T_M\_WINDOWID} \text{T_M\_RECTANGLE} \text{AXIS} \text{TEXT} \\
\text{BITMAP} \text{BUTTONBAR} \text{MESSAGE} \text{T_M\_HELP} \text{AXIS}_1 \\
\text{MENUBAR} \text{STATUS2} \text{VIEW} \text{LEGEND}
\]
TmEnumerateMethods Function

Obtains the methods that were set for a VDA Tool.

Usage

\[
\text{methods} = \text{TmEnumerateMethods} (\text{tool\_name})
\]

Parameters

\text{tool\_name} — A string containing the unique name of a VDA Tool.

Returned Value

\text{methods} — A string array containing the names of the methods that were set for the specified VDA Tool.

Example

If the TM_DISPLAY, TM_CONVERT, and TM_CODEGEN methods were set for a VDA Tool, TmEnumerateMethods returns the following:

\[
\text{PRINT, TmEnumerateMethods} (\text{tool\_name})
\]

\[
\text{TM\_DISPLAY TM\_CONVERT TM\_CODEGEN}
\]

See Also

TmExecuteMethod, TmGetMethod, TmSetMethod
TmEnumerateSelectedVars Function

Returns the names of variables on the selected variables list.

Usage

d = TmEnumerateSelectedVars()

Parameters

None.

Returned Value

d — A string array containing the $MAIN$ level names of the selected variables.

Keywords

None.

Discussion

The selected variables list is a list of variables maintained by the Tools Manager. This list enables variables to be exchanged between VDA Tools. The TmAddSelectedVars function is used to add variables to the selected variables list.

Example

This example demonstrates how variables from a VDA Tool are added to the variable selection list, exported to another VDA Tool, and deleted from the variable selection list.

PRINT, TmEnumerateSelectedVars()
; No variables are currently on the variable selection list.

PRINT, TmEnumerateVars('WzPlot_0')
HAN
; The variable HAN is currently displayed in the VDA Tool WzPlot_0.

TmAddSelectedVars, 'WzPlot_0', 'HAN'
; Add the variable HAN in the VDA Tool WzPlot to the selected variables list.

PRINT, TmEnumerateSelectedVars()
HAN
    ; Enumerate the variables that are on the selected variables list.
TmExportSelection, ['WzPlot_2']
    ; Export the variable on the selection list to another active VDA Tool.
TmDeselectVars
    ; Remove all variables from the variable selection list.

See Also
TmAddSelectedVars, TmDeselectVars, TmExport, TmExportSelection

---

**TmEnumerateToolNames Function**

Returns all the registered VDA Tool names.

**Usage**

```
names = TmEnumerateToolNames()
```

**Parameters**

None.

**Returned Value**

*names* — A 1D array containing all the names (also referred to as handles) of the currently registered VDA Tools.

**Keywords**

*Titles* — Returns a 1D string array containing the titles of the enumerated VDA Tools.

**Example**

The following code initializes the Tools Manager, gets a unique name for *WzPlot-Tool*, registers an instance of *WzPlotTool*, and enumerates all the handles currently registered with the Tools Manager.

```
TmInit
```
; Initialize the Tools Manager.
unique = TmGetUniqueToolName('WzPlotTool')
; A unique name is required for registration.
top = WwInit('VDA', 'Examples', layout, $
       Shell_name='WzPlotTool', Layout_name='toolArea', $
       Title=unique_name, /Form)
TmRegister, unique_name, top
; Register the tool.
PRINT, TmEnumerateToolNames()
   WzPlotTool_0
; Get all the handles currently registered.

See Also
TmGetTop, TmGetUniqueToolName, TmInit, TmRegister, TmUnregister

TmEnumerateVars Function
Returns all the variables associated with an instance of a VDA Tool.

Usage
variables = TmEnumerateVars(tool_name)

Parameters

   tool_name — A string containing the unique name of a VDA Tool.

Returned Value
variables — A 1D string array containing the names of the variables associated
with the specified VDA Tool.

Discussion
This function is used frequently in VDA Tool programs to extract variable names
that were passed to a VDA Tool for use in plotting, or data selection, or other kinds
of routines that require variables. For example, in a graphics tool,
TmEnumerateVars might be called by a plotting routine to find the names of variables that were passed to the VDA Tool program.

**Example**

The following code fragment shows how TmEnumerateVars can be used to get the names of variables from a VDA Tool, and then use UPVAR to bind the variable locally, in the WzTemplateConvert procedure.

```plaintext
PRO WzTemplateConvert, tool_name, user_data=user_data
   plot_var = TmEnumerateVars(tool_name)
   UPVAR, plot_var(0), local
   . . .
```

**See Also**

TmAddVar, TmDelVar, TmGetVarMainName

---

**TmExecuteMethod Procedure**

Executes a method that was set by TmSetMethod.

**Usage**

TmExecuteMethod, *tool_name, method_name*

**Parameters**

*tool_name* — The name of a VDA Tool.

*method_name* — The name of the method to execute.

**Keywords**

*No_Wset* — If nonzero, indicates that the WSET command is not run to change the current window.

*Top* — The name of the top-level widget.
Discussion

TmExecuteMethod is used in a VDA Tool program to tell the VDA Tools Manager when to execute a method for a particular instance of a VDA Tool. Methods allow multiple instances of VDA Tools to take actions independently, such as displaying graphics or generating PV-WAVE code. When a method is activated, a method procedure is called that performs an action for the specific instance of the VDA Tool that triggered the method. It is up to the VDA Tools developer to determine which events trigger specific methods, and to write appropriate method call procedures.

Example

The following method call procedure contains TmExecuteMethod statements to convert coordinates and redraw graphics.

```pro
PRO DrawPlotCB, wid, data
   ; Get the tool name.
   x = WtGet( wid, /Parent)
   tool_name = WtGet(x, /Userdata)
   TmExecuteMethod, tool_name, 'TM_CONVERT'
   ; Convert the coordinates.
   TmExecuteMethod, tool_name, 'TM_DISPLAY'
   ; Redraw the graphics to ensure everything is ok.
END
```

See Also

TmEnumerateMethods, TmGetMethod, TmSetMethod
**TmExport Procedure**

Exports $MAIN$-level variables to specified VDA Tools or to all currently active VDA Tools.

**Usage**

TmExport, *variable_names, destination_tool_names*

**Parameters**

*variable_names* — A string array containing the names of $MAIN$ level variables to export.

*destination_tool_names* — A string array containing the unique names of currently active VDA Tools to export the data to.

**Keywords**

*All* — If present and nonzero, exports the named variables to all VDA Tools that accept imported data.

**Discussion**

The UPVAR or INFO commands can be used to obtain the names of variables on the $MAIN$ program level.

**Example**

The following commands export two $MAIN$ level variables to the VDA Tool WzPlot_1. The VDA Tool WzPlot_1 must be running before the data is exported to it.

```wave
WAVE> info, /var
    HIN    FLOAT    = Array(100)
    HAN    FLOAT    = Array(100)
TmExport, ['han', 'hin'], ['WzPlot_1']
```

**See Also**

INFO (in the PV-WAVE Reference), UPVAR (in the PV-WAVE Reference), TmExportSelection
**TmExportSelection Procedure**

Exports the contents of the variable selection list to specified VDA Tools.

**Usage**

TmExportSelection, *destination_tool_names*

**Parameters**

*destination_tool_names* — A string array containing the names of currently active VDA Tools to export the data to.

**Keywords**

*All* — If present and nonzero, export the variables on the selection list to all VDA Tools that accept imported data.

**Discussion**

The Tools Manager maintains a list of variables that have been selected in a VDA Tool. The selected variables list enables variables to be exchanged between VDA Tools. Variables are added to the list with the TmAddSelectedVars routine.

**Example**

This example demonstrates how variables from a VDA Tool are added to the variable selection list, exported to another VDA Tool, and deleted from the variable selection list.

```plaintext
PRINT, TmEnumerateSelectedVars()
; No variables are currently on the variable selection list.

PRINT, TmEnumerateVars('WzPlot_0')
HAN
; The variable HAN is currently displayed in the VDA Tool WzPlot_0.

TmAddSelectedVars, 'WzPlot_0', 'HAN'
; Add the variable HAN in the VDA Tool WzPlot to the selected variables list.

PRINT, TmEnumerateSelectedVars()
HAN
; Enumerate the variables that are on the selected variables list.
```
TmExportSelection, ['WzPlot_2']
   ; Export the variable on the selection list to another active VDA Tool.
TmDeselectVars
   ; Remove all variables from the variable selection list.

See Also
TmAddSelectedVars, TmDeselectVars, TmEnumerateSelectedVars, TmExport

---

**TmGetAttribute Function**

Returns the value that was set for an attribute in a VDA Tool instance.

**Usage**

\[
value = \text{TmGetAttribute}(\text{tool\_name}, \text{item}, \text{attr\_name})
\]

**Parameters**

- **tool\_name** — A string containing the unique name of a VDA Tool.
- **item** — A string containing the name of a graphical element, variable, or other item in the VDA Tool.
- **attr\_name** — A string containing an attribute name to get for the given *item*.

**Returned Value**

- **value** — The value that was set for the attribute *attr\_name* in the VDA Tool instance.

**Keywords**

- **Default** — Used to set a default value for the attribute. If the attribute you want to get was not set, then the function will return this default value.

**Discussion**

TmGetAttribute is typically used in a method call procedure. For instance, when the drawing method TM_DISPLAY is activated for a VDA Tool, a procedure is
executed that performs some kind of graphics operation. TmGetAttribute allows the procedures to obtain information from the Tools Manager about the values it needs to render the plot.

Example 1

Calls like the following can be used by a VDA Tool plotting routine to obtain the line color and linestyle that were registered with the Tools Manager using TmSetAttribute. In this case, the attributes are associated with a variable.

```plaintext
color = TmGetAttribute(tool_name, var, 'COLOR')
lstyle= TmGetAttribute(tool_name, var, 'LSTYLE')
```

Example 2

The title for the VDA Tool could be kept in an attribute, and the color of a variable could be set for future usage:

```plaintext
old_title = TmSetAttribute(tool_name, 'TM_TITLE', 'TEXT', $ title_text)
old_color = TmSetAttribute(tool_name, 'VARIABLE1', 'COLOR', 'RED')
```

In the WzPlotTool plotting routine, the tool asks for some attribute settings:

```plaintext
xminrange = TmGetAttribute(tool_name, 'RANGE', 'X_MIN') ; Gets the Xrange for image.
ymaxrange = TmGetAttribute(tool_name, 'RANGE', 'Y_MAX') ; Gets the Yrange.
z_rotat = TmGetAttribute(tool_name, 'ROTATION', 'Z') ; Gets the Z_Rotation.
```

Example 3

This example code demonstrates the effect of the Default keyword when a graphical element is not set and when a one is set.

```plaintext
INFO, TmGetAttribute(n0, 'TM', 'BAD_ATTR')
<Expression> STRING = '' ; The attribute was not previously set.
INFO, TmGetAttribute(n0, 'TM', 'BAD_ATTR', default=1B)
<Expression> BYTE = 1 ; The attribute is set with the Default keyword.
```
INFO, TmGetAttribute(n0, 'TM', 'BACKGROUND')
<Expression>    INT       =       90
    ; The attribute BACKGROUND is already set.
INFO, TmGetAttribute(n0, 'TM', 'BACKGROUND', default=1B)
<Expression>    INT       =       90
    ; The Default keyword has no effect if the attribute was already set.

See Also
TmEnumerateAttributes, TmSetAttribute

---

**TmGetMessage Function**

 Loads a string resource file into the resource database and extracts a message string from the database.

**Usage**

```plaintext
message = TmGetMessage( [message_file], message_code)
```

**Parameters**

- `message_file` — (optional) The name of a string resource file where the messages are stored. If this file has not already been loaded, it will be loaded before the message is retrieved.
- `message_code` — A string used to identify a message in a string resource file.

**Returned Value**

- `message` — A string containing the message that was extracted from the string resource file.

**Keywords**

None.
**Discussion**

You only need to supply the filename for the *message_file* input parameter. The pathname is assumed to be the following, where !Lang represents the value of the !Lang system variable in PV-WAVE:

(UNIX)  \(<\text{wavedir}\>/\text{xres/!Lang/vdatools}\)

(OpenVMS)  \(<\text{wavedir}>:\text{[XRES.!'Lang.VDATOOLS]}\)

(Windows)  \(<\text{wavedir}\>\text{xres/!Lang\vdatools}\)

Where <wavedir> is the main PV-WAVE directory.

Although the message file is passed to each call of TmGetMessage, the specified string is only loaded once.

**Example**

This example shows a portion of the string resource file for the WzPlot VDA Tool: wzglobal.ads. An excerpt from the source code for the WzPlot VDA Tool shows how TmGetMessage is used to load this file into the resource database and extract a string from the database. In this example, the string is extracted and printed in the Message Area of the WzPlot window whenever a new variable is passed to WzPlot.

**String resource file excerpt:**

```
!       Data Import Messages (in wzglobal.ads)
!
   imported_var:   Imported Variable:
```

**WzPlot source code:**

```
msg = TmGetMessage('wzglobal.ads', 'imported_var')
; Get the appropriate message text from the string resource file.
msg = msg + variable_list(j)
; Build a message string with the variable name.
WoAddMessage, tool_name, Message=msg
; Display the message in the VDA Tool's Message Area.
```

**See Also**

WoAddMessage
**TmGetMethod Function**

Returns the data structure of the specified method.

**Usage**

\[
\text{method\_call} = \text{TmGetMethod(}\text{tool\_name, method\_name})
\]

**Parameters**

- **tool\_name** — A string containing the unique name of a VDA Tool.
- **method\_name** — A string containing the name of a method.

**Returned Value**

- **method\_call** — The name of the procedure in which the method was set.

**Discussion**

The method data structure contains two tags: the name of the method procedure and a variable for user data.

**Example**

The following command shows that a method procedure called WzPlotDisplay is registered with the TM_DISPLAY method in the VDA Tool instance WzPlot_0.

```plaintext
PRINT, TmGetMethod(‘WzPlot_0’, ‘TM_DISPLAY’)
{ WzPlotDisplay{}}
```

**See Also**

- TmEnumerateMethods, TmExecuteMethod, TmSetMethod
**TmGetTop Function**

Gets the top-level widget ID for a VDA Tool.

**Usage**

\[ top = TmGetTop(tool\_name) \]

**Parameters**

- **tool\_name** — A string containing the unique name of a VDA Tool.

**Returned Value**

- **top** — (long) The top-level widget ID for the VDA Tool.

**Keywords**

None.

**Discussion**

TmGetTop is used to obtain the widget ID of the top-level VDA Tool widget. This function is called when you exit a VDA Tool, unregister a VDA Tool, and create a dialog box (dialog boxes are children of the top-level VDA Tool widget).

**Example**

This example shows how the top-level widget ID is used to create a dialog box. The dialog box is actually a child of the top-level VDA Tool widget.

```dialect
parent = TmGetTop(tool\_name) ; Get the top-level widget ID.

title = TmGetMessage('wzplot.ads', 'data\_export\_title') $ + ' ' + tool\_name ; Get message text from the string resource file.

dialog = WoGenericDialog(parent, layout, $ 'WzPlotDataExportCB', Dialog\_name = 'exportDataDialog', $ Buttons=buttons, Title=title, $ Help=['Export Selected Data Dialog Box', helpfile], $ /Ok, /Apply, /Cancel)
; Create the dialog box using the top-level widget ID obtained from TmGetTop.
```
TmGetUniqueToolName Function

Returns a unique name for a particular instance of a specified VDA Tool.

Usage

unique_name = TmGetUniqueToolName(tool_name)

Parameters

tool_name — A string containing the unique name of a VDA Tool.

Returned Value

unique_name — A unique name for the specified VDA Tool.

Keywords

None.

Discussion

Each VDA Tool must have a unique name, which allows the Tools Manager to keep track of multiple instances of a VDA Tool. This unique name must be registered with the Tools Manager before the VDA Tool can be used. To register a VDA Tool name, use the TmRegister function.

Example

This call obtains a unique name for the VDA Tool called WzPlotTool.

unique = TmGetUniqueToolName('WzPlotTool')
See Also

TmEnumerateToolNames, TmGetMessage, TmGetTop, TmInit, TmRegister, TmUnregister

---

**TmGetVarMainName Function**

Returns the $MAIN$ level name of a variable.

**Usage**

```python
var_name = TmGetVarMainName(tool_name, local_variable)
```

**Parameters**

- `tool_name` — A string containing the unique name of a VDA Tool.
- `local_variable` — The local variable for which you want the $MAIN$ level name.

**Returned Value**

- `var_name` — The $MAIN$ level name of the local variable.

**Keywords**

- `NoCreate` — If nonzero, the function does not create a new $MAIN$ variable, but just returns the variable name.
- `Root` — A string containing the base for the name of the $MAIN$ variable.

**Discussion**

If the variable does not exist on the $MAIN$ level, a new $MAIN$ level variable is created, and its name is returned.

**See Also**

TmAddVar, TmDelVar, TmEnumerateVars
**TmInit Procedure**

Initializes the VDA Tools Manager layer.

**Usage**

TmInit

**Parameters**

None.

**Keywords**

None.

**Discussion**

TmInit sets up the data structures, initializes the currently selected variables, and performs other setup functions. If TmInit has already been called in the current PV-WAVE session, subsequent calls to TmInit are ignored.

**Example**

You must call TmInit before using any VDA Tools Manager routines.

TmInit

**See Also**

TmEnumerateToolNames, TmGetMessage, TmGetTop, TmGetUniqueToolName, TmRegister, TmUnregister
**TmList Function**

Creates a list item.

**Usage**

```python
list = TmList(tool_name)
```

**Input Parameters**

*tool_name* — A string specifying the unique name of a VDA Tool.

**Returned Value**

*list* — A string containing a unique name for the list. This value is used as input to the other list functions.

**Keywords**

*ExtendSize* — An integer specifying the number of items by which to extend the list. (Default: 10)

**Discussion**

This routine creates a Tm item with the name LIST_# (where # is a number assigned to ensure that the list name is unique) with attributes ITEMS and FREE.

**Example**

This example creates a list associated with *tool_name*. As items are added and the list becomes full, the list is extended by 25 items.

```python
list_name = TmList(tool_name, ExtendSize=25)
INFO, list_name
   LIST_NAME       STRING     = 'LIST_0'
```

**See Also**

TmListAppend, TmListClear, TmListDelete, TmListDestroy, TmListExtend, TmListGetMethod, TmListInsert, TmListReplace, TmListRetrieve, TmListSetMethod
**TmListAppend Procedure**

Adds a new item at the end of the specified list.

**Usage**

TmListAppend, `tool_name`, `list_name`, `item`

**Input Parameters**

- `tool_name` — A string specifying the unique name of a VDA Tool.
- `list_name` — A string specifying the unique name of a list.
- `item` — The item to append to the list. This parameter can be any PV-WAVE variable.

**Keywords**

None.

**Discussion**

If a new item is required elsewhere in the list, use TmListInsert instead of TmListAppend. If the list is not long enough to accept another item, the list is extended by the number of items specified when the list was created.

This procedure defines the behavior of the TM_LIST_APPEND method.

**Example**

The following code adds a string and an array to the end of a list.

```plaintext
list_name = TmList(tool_name)
TmListAppend, tool_name, list_name, 'My String'
INFO, TmListRetrieve(tool_name, list_name), /Full
<Expression> LIST      = List(1)
<Expression> STRING    = 'My String'
; Adds the string 'My String' at the end of the list.
TmListAppend, tool_name, list_name, [1,2,3]
; Adds the array [1,2,3] at the end of the list.
```
INFO, TmListRetrieve(tool_name, list_name), /Full
<Expression> LIST = List(2)
STRING = 'My String'
INT = Array(3)

See Also
TmList, TmListSetMethod, TmListInsert

**TmListClear Procedure**

Resets a specified list to its initial state, clearing all previously defined items.

**Usage**

TmListClear, tool_name, list_name

**Input Parameters**

*tool_name* — A string specifying the unique name of a VDA Tool.
*list_name* — A string specifying the unique name of a list.

**Keywords**

None.

**Discussion**

This procedure defines the behavior of the TM_LIST_CLEAR method. TmListDelete can be used to delete individual items in the list.

**Example**

The following code creates a list and then clears all items from the list.

```plaintext
list_name = TmList(tool_name)
TmListAppend, tool_name, list_name, 'My String'
INFO, TmListRetrieve(tool_name, list_name), /Full
<Expression> LIST = List(1)
STRING = 'My String'
TmListClear, tool_name, list_name
```
INFO, TmListRetrieve(tool_name, list_name), /Full
<Expression> LIST = List(0)

See Also
TmList, TmListSetMethod, TmListDelete

TmListDelete Procedure

Deletes an item in the specified list.

Usage
TmListDelete, tool_name, list_name [, pos]

Input Parameters

tool_name — A string specifying the unique name of a VDA Tool.
list_name — A string specifying the unique name of a list.

pos — (optional) An integer specifying the position in the list of the item to delete. The first item is 0, the second item is 1, and so on. If pos is not specified, the last item in the list is deleted.

Keywords
None.

Discussion
This procedure defines the behavior of the TM_LIST_DELETE method.

Example
The following code creates a list, removes the last element in the list, and then removes the first item in the list.

; Removes the last element in the list.
list_name = TmList(tool_name)
TmListAppend, tool_name, list_name, ‘First Item’
TmListAppend, tool_name, list_name, 2L
TmListAppend, tool_name, list_name, 3.3
INFO, TmListRetrieve(tool_name, list_name), /Full
<Expression> LIST = List(3)
STRING = 'First Item'
LONG = 2
FLOAT = 3.30000

TmListDelete, tool_name, list_name
INFO, TmListRetrieve(tool_name, list_name), /Full
<Expression> LIST = List(2)
STRING = 'First Item'
LONG = 2

; Removes the first item in the list.
TmListDelete, tool_name, list_name, 0
INFO, TmListRetrieve(tool_name, list_name), /Full
<Expression> LIST = List(1)
LONG = 2

See Also
TmList, TmListSetMethod, TmListClear

---

**TmListDestroy Procedure**

Clears all items and destroys the list.

**Usage**

TmListDestroy, tool_name, list_name

**Input Parameters**

*tool_name* — A string specifying the unique name of a VDA Tool.

*list_name* — A string specifying the unique name of a list.

**Keywords**

None.
**Discussion**

Once a list is destroyed, do not access the list instance again without creating it using the TmList routine.

This procedure defines the behavior of the TM_LIST_DESTROY method.

**Example**

These commands destroy a list so that list_name is no longer a valid list handle.

```plaintext
list_name = TmList(tool_name)
TmListAppend, tool_name, list_name, 'First Item'
TmListDestroy, tool_name, list_name
```

**See Also**

TmList, TmListSetMethod, TmListClear

---

**TmListExtend Procedure**

Extends the specified list by adding empty items.

**Usage**

TmListExtend, tool_name, list

**Input Parameters**

- `tool_name` — A string specifying the unique name of a VDA Tool.
- `list_name` — A string specifying the unique name of a list.

**Keywords**

- `NumItems` — An integer specifying the number of items to add. The value of this keyword overrides the ExtendSize keyword specified in TmList.

**Discussion**

The list is extended in increments greater than 1 to avoid excessive copying.
This procedure defines the behavior of the TM_LIST_EXTEND method.

**Example**

These commands extend a list.

```
; Extends the list by the number specified in the ExtendSize
; keyword of the TmList command (default: 10).
TmListExtend, tool_name, list_name

; Extends the list by 15 items.
TmListExtend, tool_name, list_name, NumItems=15
```

**See Also**

TmList, TmListSetMethod

---

**TmListGetMethod Function**

Returns the procedure name associated with the specified list method name.

**Usage**

```
procedure_name = TmListGetMethod(tool_name, list_name, method_name)
```

**Input Parameters**

- **tool_name** — A string specifying the unique name of a VDA Tool.
- **list_name** — A string specifying the unique name of a list.
- **method_name** — A string specifying the name of a method.

**Returned Value**

```
procedure_name — A string containing the name of the list method procedure if
    set, otherwise a NULL string is returned.
```
Keywords
None.

Discussion
The default list method names and their procedures are listed in the following table:

<table>
<thead>
<tr>
<th>Description</th>
<th>Method Name</th>
<th>Method Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extend a list</td>
<td>TM_LIST_EXTEND</td>
<td>TmListExtend</td>
</tr>
<tr>
<td>Append to a list</td>
<td>TM_LIST_APPEND</td>
<td>TmListAppend</td>
</tr>
<tr>
<td>Insert in a list</td>
<td>TM_LIST_INSERT</td>
<td>TmListInsert</td>
</tr>
<tr>
<td>Retrieve from a list</td>
<td>TM_LIST_RETRIEVE</td>
<td>TmListRetrieve</td>
</tr>
<tr>
<td>Replace a list element</td>
<td>TM_LIST_REPLACE</td>
<td>TmListReplace</td>
</tr>
<tr>
<td>Delete (Undo) previous change to a list</td>
<td>TM_LIST_DELETE</td>
<td>TmListDelete</td>
</tr>
<tr>
<td>Clear list elements</td>
<td>TM_LIST_CLEAR</td>
<td>TmListClear</td>
</tr>
<tr>
<td>Destroy list elements</td>
<td>TM_LIST_DESTROY</td>
<td>TmListDestroy</td>
</tr>
</tbody>
</table>

**NOTE** Code for the default list method procedures are contained in the file, where WAVE_DIR is the the main PV-WAVE directory.

(UNIX) WAVE_DIR/lib/vdatools/tmlist.pro

(OpenVMS) WAVE_DIR:[LIB.VDATOOLS]TMLIST.PRO

(Windows) WAVE_DIR\lib\vdatools\tmlist.pro

Use TmListSetMethod to override the default method procedures.

Example
In this example, a user-defined TM_LIST_APPEND method is called. The method adds all items except the string ‘Dont_Add’.

```
; Create the method procedure.
PRO MyListAppend, tool_name, list_name, item
   DECLARE FUNC, TmGetAttribute
```

TmListGetMethod Function 453
DECLARE FUNC, TmSetAttribute

items = TmGetAttribute(tool_name, list_name, 'ITEMS')
free = TmGetAttribute(tool_name, list_name, 'FREE')

IF free GE N_ELEMENTS(items) THEN BEGIN
  TmListExtend, tool_name, list_name
  items = TmGetAttribute(tool_name, list_name, 'ITEMS')
ENDIF
IF item NE 'Dont_Add' THEN BEGIN
  items(free) = item
  s = TmSetAttribute(tool_name, list_name, 'ITEMS', items)
  s = TmSetAttribute(tool_name, list_name, 'FREE', free+1)
ENDIF
END

; Create a list and append to it.
list_name = TmList(tool_name)
TmListSetMethod, tool_name, list_name, 'TM_LIST_APPEND', $
  'MyListAppend'
TmListAppend, tool_name, list_name, 'First Item'
INFO, TmListRetrieve(tool_name, list_name), /Full
  <Expression>    LIST    = List(1)
  STRING        = 'First Item'
TmListAppend, tool_name, list_name, 'Dont_Add'
INFO, TmListRetrieve(tool_name, list_name), /Full
  <Expression>    LIST    = List(1)
  STRING        = 'First Item'

; Use TmListGetMethod to access the name of the method and
; execute it explicitly.
new_var = 'Second Item'
var_name = 'new_var'
method = TmListGetMethod(tool_name, list_name, 'TM_LIST_APPEND')
INFO, method
  METHOD          STRING    = 'MyListAppend'
s = EXECUTE(method + ',', tool_name, list_name, ' ' + var_name)
**TmListInsert Procedure**

Inserts a new item into the specified list.

**Usage**

TmListInsert, `tool_name`, `list_name`, `item`, `pos`

**Input Parameters**

- `tool_name` — A string specifying the unique name of a VDA Tool.
- `list_name` — A string specifying the unique name of a list.
- `item` — A string specifying the item to append to the list.
- `pos` — An integer specifying the position before which to insert a new item into the list.

**Keywords**

None.

**Discussion**

The `pos` parameter indicates the position before which to insert the new item. The first item is 0; the second item is 1, and so on. If a new item is required at the end of the list, use TmListAppend instead of TmListInsert.

This procedure defines the behavior of the TM_LIST_INSERT method.
Example

This example creates a list and inserts items into it.

; Inserts the string 'New Item' before the first item in the list.
list_name = TmList(tool_name)
TmListAppend, tool_name, list_name, 'First Item'
TmListAppend, tool_name, list_name, 2L
TmListInsert, tool_name, list_name, 'New Item', 0
INFO, TmListRetrieve(tool_name, list_name), /Full
<Expression>    LIST      = List(3)
STRING    = 'New Item'
STRING    = 'First Item'
LONG      =            2

; Insert the float 1.1 before the third item in the list.

TmListInsert, tool_name, list_name, 1.1, 2
INFO, TmListRetrieve(tool_name, list_name), /Full
<Expression>    LIST      = List(4)
STRING    = 'New Item'
STRING    = 'First Item'
FLOAT     =       1.10000
LONG      =            2

See Also
TmList, TmListSetMethod, TmListAppend

TmListReplace Procedure

Replaces an item in a list with a new item.

Usage

TmListReplace, tool_name, list_name, item, pos

Input Parameters

tool_name — A string specifying the unique name of a VDA Tool.
list_name — A string specifying the unique name of a list.

item — The new item to add to the list. This parameter may be any PV-WAVE variable.

pos — An integer specifying the position of the item to replace.

**Keywords**

None.

**Discussion**

The pos parameter indicates the position of the item to replace. The first item is 0; the second item is 1, and so on. If the position specified falls beyond the end of the list, the item will be appended to the list.

This procedure defines the behavior of the TM_LIST_REPLACE method.

**Example**

This example replaces the third item in the list with the string ’New Item’.

```
list_name = TmList(tool_name)
TmListAppend, tool_name, list_name, 'First Item'
TmListAppend, tool_name, list_name, 2L
TmListAppend, tool_name, list_name, 3.3
INFO, TmListRetrieve(tool_name, list_name), /Full
<Expression>    LIST      = List(3)
STRING    = 'First Item'
LONG      =            2
FLOAT     =       3.30000
TmListReplace, tool_name, list_name, 'New Item', 2
INFO, TmListRetrieve(tool_name, list_name), /Full
<Expression>    LIST      = List(3)
STRING    = 'First Item'
LONG      =            2
STRING    = 'New Item'
```

**See Also**

TmList, TmListSetMethod
**TmListRetrieve Function**

Gets the items currently set in the specified list.

**Usage**

\[ \text{items} = \text{TmListRetrieve}(tool\_name, list\_name) \]

**Input Parameters**

- **tool\_name** — A string specifying the unique name of a VDA Tool.
- **list\_name** — A string specifying the unique name of a list.

**Returned Value**

- **items** — A list array containing the currently set items or an empty list if no items have been set.

**Keywords**

- **Count** — Returns the number of items retrieved. (integer)

**Discussion**

Only those items that have been set are returned. Items that have been allocated but not used are not returned.

This procedure defines the behavior of the TM_LIST_RETRIEVE method.

**Example**

This example retrieves all the items in a list and uses the **Count** keyword to get the number of items in the list.

```plaintext
; Retrieve all the items in the list.
list\_name = TmList(tool\_name)
TmListAppend, tool\_name, list\_name, ‘First Item’
TmListAppend, tool\_name, list\_name, 2L
TmListInsert, tool\_name, list\_name, ‘New Item’, 0
```
items = TmListRetrieve(tool_name, list_name)
INFO, items, /Full
   ITEMS       LIST  = List(3)
   STRING    = 'New Item'
   STRING    = 'First Item'
   LONG      = 2

; Use the Count keyword to return the number of items in the list.
items = TmListRetrieve(tool_name, list_name, Count=count)
INFO, items
   ITEMS       LIST  = List(3)
PRINT, count
   3

See Also
TmList, TmListSetMethod

TmListSetMethod Procedure

Sets the method procedure name for a specific list method.

Usage
TmListSetMethod, tool_name, list_name, method_name, method

Input Parameters

    tool_name  — A string specifying the unique name of a VDA Tool.
    list_name  — A string specifying the unique name of a list.
    method_name — A string specifying the name of the method to override.
    method  — A string specifying the name of the procedure to execute when
               method_name is requested.

Keywords
None.
Discussion

This routine overrides the default method procedure provided for the following list methods:

<table>
<thead>
<tr>
<th>Description</th>
<th>Method Name</th>
<th>Method Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extend a list</td>
<td>TM_LIST_EXTEND</td>
<td>TmListExtend</td>
</tr>
<tr>
<td>Append to a list</td>
<td>TM_LIST_APPEND</td>
<td>TmListAppend</td>
</tr>
<tr>
<td>Insert in a list</td>
<td>TM_LIST_INSERT</td>
<td>TmListInsert</td>
</tr>
<tr>
<td>Retrieve from a list</td>
<td>TM_LIST_RETRIEVE</td>
<td>TmListRetrieve</td>
</tr>
<tr>
<td>Replace a list element</td>
<td>TM_LIST_REPLACE</td>
<td>TmListReplace</td>
</tr>
<tr>
<td>Delete (Undo) previous change to a list</td>
<td>TM_LIST_DELETE</td>
<td>TmListDelete</td>
</tr>
<tr>
<td>Clear list elements</td>
<td>TM_LIST_CLEAR</td>
<td>TmListClear</td>
</tr>
<tr>
<td>Destroy list elements</td>
<td>TM_LIST_DESTROY</td>
<td>TmListDestroy</td>
</tr>
</tbody>
</table>

**NOTE** Code for the default list method procedures are contained in the file, where **WAVE_DIR** is the main PV-WAVE directory.

(UNIX)  WAVE_DIR/lib/vdatools/tmlist.pro

(OpenVMS)  WAVE_DIR:[LIB.VDATOOLS]TMLIST.PRO

(Windows)  WAVE_DIR\lib\vdatools\tmlist.pro

When overriding a method, you are responsible for properly updating the associated attributes to reflect the desired change.

Example

See the example for TmListGetMethod.

See Also

TmList, TmListGetMethod
**TmPaste Procedure**

Pastes the graphical elements from the clipboard to the specified VDA Tool.

**Usage**

TmPaste, *tool_name*

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool into which to paste the graphical elements.

**Keywords**

None.

**Discussion**

The TmAddSelectedGrael function is used to add graphical elements to a selection list maintained by the Tools Manager. Once on the selection list, graphical elements can be copied, pasted, cut, or deleted. TmPaste pastes the graphical element or elements currently on the clipboard (a temporary buffer) to the specified VDA Tool.

**Example**

The following commands obtain the names of the graphical elements registered for a VDA Tool, add one of the graphical elements to the selection list, then copy and paste the graphical element (a rectangle).

```plaintext
PRINT, TmEnumerateGraels(‘WzPlot_0’)

TM_WINDOWID TM RECTANGLE LINE AXIS LEGEND TEXT MENUBAR
    BUTTONBAR MESSAGE XX TM_HELP AXIS_0 AXIS_1 TM_DRAWING
    RECTANGLE_0 LINE_1 GROUP_0
TmAddSelectedGrael, ‘WzPlot_0’, ‘RECTANGLE_0’
PRINT, TmEnumerateSelectedGraels(‘WzPlot_0’)
    RECTANGLE_0
TmCopy, ‘WzPlot_0’
TmPaste, ‘WzPlot_0’
```
See Also
TmCopy, TmCut, TmDelete

TmRegister Procedure

Registers a VDA Tool with the Tools Manager.

Usage
TmRegister, unique_name, topShell

Parameters
unique_name — The name returned by TmGetUniqueToolName.
topShell — The widget ID of the top-level shell of the VDA Tool.

Keywords
Title — A string specifying the title of the VDA Tool. This title is automatically added to the Windows menu.

Discussion
You have to register a VDA Tool with the Tools Manager before any operations can be performed with the Tool. By registering the Tool, you add it to the Tools Manager data structure, which enables the Tools Manager to keep track of the characteristics of multiple instances of the Tool.

The VDA Tool name that is registered must be unique. An error results if the input VDA Tool name already exists. Use TmGetUniqueToolName to obtain a unique name.

If you do not set the window title with the Title keyword, then you have to use WoSetWindowTitle to add the title to the Windows menu.

Example
The following lines show how a unique VDA Tool name is obtained and registered with the Tools Manager. Note that the top-level widget must be defined before the VDA Tool is registered.
TmInit
unique = TmGetUniqueToolName('WzPlotTool')
    ; Obtain a unique name for the VDA Tool.
top = WwInit('VDA', 'Examples', layout, $
    Shell_name='wzplottool', Layout_name='toolArea', $
    Title=unique_name, /Form)
    ; Initialize the top-level widget.
TmRegister, unique, top
    ; Register the unique VDA Tool name with the Tools Manager.

The remainder of the program is omitted from this example.

See Also
TmEnumerateToolNames, TmGetMessage, TmGetTop,
TmGetUniqueToolName, TmInit, TmUnregister, WoSetWindowTitle

---

**TmRestoreTemplate Function**

Restores a saved VDA Tool template.

**Usage**

\[
status = TmRestoreTemplate(tool_name, filename)
\]

**Parameters**

- `tool_name` — A string containing the unique name of a VDA Tool.
- `filename` — A string containing the name of the file containing the saved template.

**Keywords**

None.

**Returned Value**

- `status` — A returned value of 1 indicates success; 0 (zero) indicates failure.
Discussion

A template is a VDA Tool without any data associated with it. The template contains all of the modifications to the VDA Tool — colors, axes, graphical elements — that were set when the template file was saved.

When a VDA Tool is saved with TmSaveTools, it can be restored in two ways: using TmRestoreTools or TmRestoreTemplate. TmRestoreTools restores the entire VDA Tool, including the data associated with it. TmRestoreTemplate restores the VDA Tool, but it does not restore any data that was associated with the VDA Tool when it was saved.

Example

```plaintext
status=TmRestoreTemplate(tool_name, 'saved_template.xdr')
```

See Also

TmRestoreTools, TmSaveTools

TmRestoreTools Function

Restores the VDA Tools that were saved with the TmSaveTools procedure.

Usage

```plaintext
status = TmRestoreTools(filename)
```

Parameters

- **filename** — A string containing the name of the VDA Tool save file created with the TmSaveTools procedure.

Keywords

- **Restore_CT** — If present and nonzero, restores the colortable from a save file created by TmSaveTools. If the save file does not contain a colortable, then this keyword is ignored.

- **Template** — If present and nonzero, saved VDA Tools are restored without the values of their variables.
**Returned Value**

*status* — A returned value of 1 indicates success; 0 (zero) indicates failure.

**Discussion**

This procedure is designed specifically for use in restoring VDA Tools that were saved with TmSaveTools. It differs from the regular RESTORE procedure in that it ensures that the Tools Manager data structure is restored properly.

**Example**

```
status=TmRestoreTools('vdaapp.sav')
```

**See Also**

*TmSaveTools, TmRestoreTemplate*

---

**TmSaveTools Procedure**

Saves the specified VDA Tools in a file.

**Usage**

```
TmSaveTools, filename [, tool_names]
```

**Parameters**

*filename* — A string containing the name of a file in which to save the specified VDA Tools.

*tool_names* — (optional) A string array of unique VDA Tool names.

**Keywords**

*All* — If present and nonzero, all currently running tools are saved in the specified file. This keyword takes precedence over the *tool_names* parameter.
**Discussion**

This procedure is designed specifically for use in saving VDA Tools. It differs from the regular SAVE procedure in that it ensures that the Tools Manager data structure is saved properly so that it can be restored later.

If neither the `tool_names` parameter nor the `All` keyword is specified, no action is taken.

**Example**

```
TmSaveTools, 'vdaapp.sav', /All
```

**See Also**

`TmRestoreTools`, `TmRestoreTemplate`

---

**TmSetAttribute Function**

Set an attribute for an item in the given VDA Tool.

**Usage**

```
value = TmSetAttribute(tool_name, item, attr_name, attr_value)
```

**Parameters**

- `tool_name` — A string containing the unique name of a VDA Tool.
- `item` — The name of an item associated with the VDA Tool.
- `attr_name` — A string specifying an attribute to set for the given `item`.
- `attr_value` — A value to set for the given attribute.

**Returned Value**

- `value` — Returns the previously set attribute value. If no attribute value was previously set, the function returns an empty string.

**Keywords**

None.
Discussion

TmSetAttribute is used to register attributes and their values with the Tools Manager. The Tools Manager stores information about every item added to a VDA Tool instance in its data structure. This information includes the VDA Tool’s unique name and items that have been assigned to the VDA Tool, such as graphical elements and variables. Each item added to a VDA Tool can have a set of characteristics that the VDA Tool programmer defines. These characteristics are called attributes, and attributes can be assigned values.

Example

Assume that an instance of a VDA Tool is currently running, and that the variable var was passed to the VDA Tool (possibly when the VDA Tool was called). The following code retrieves the name of the variable that was passed to the VDA Tool and registers that variable with the Tools Manager. Then, several attributes are set for the variable for that instance of the VDA Tool. When the VDA Tool displays the data in the variable, the plot will reflect these set attributes (e.g., line color, line-style, plot symbol type, and symbol color).

```
INFO, var, upvar=main_name
   ; From within the VDA Tool program level, obtain the name of the
   ; variable var on the $MAIN$ program level.
TmAddVar, unique, main_name
   ; Register the variable to the Tools Manager.

tmp = TmSetAttribute(unique, main_name, ‘COLOR’, 6)
   ; Set attributes with which to display the variable: color, linestyle, plot symbol,
   ; and symbol color.

tmp = TmSetAttribute(unique, main_name, ‘LSTYLE’, 8)
tmp = TmSetAttribute(unique, main_name, ‘PSYM’, 10)
tmp = TmSetAttribute(unique, main_name, ‘PSYM_COLOR’, 12)
```

See Also

TmEnumerateAttributes, TmGetAttribute
**TmSetMethod Procedure**

Sets a method for a given VDA Tool.

**Usage**

TmSetMethod, *tool_name, method_name, method_call*

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool.

*method_name* — The name of the method to set for the given VDA Tool.

*method_call* — The name of a procedure to execute when the method is called.

**Keywords**

*user_data* — A string containing data associated with the method.

**Discussion**

A method is a program that is executed in response to a trigger in a VDA Tool, such as a menu or button selection or mouse click. Normally, a VDA Tool will have several methods defined for it.

When a method is executed, the Tools Manager helps direct the subsequent action. The Tools Manager keeps track of the unique instance of the VDA Tool to which to apply the method, the name of the program to execute, and any data associated with that specific tool that the method program needs.

**Method Call Procedure Parameters**

The method call procedure must have the following parameters:

*tool_name* — The name of the VDA Tool for which the method is set.

*userdata* — Any data you wish to pass to the procedure.

**Example**

The following TmSetMethod calls set methods for drawing, code generation, restore, and coordinate conversion. The function takes the unique name of the VDA
Tool, the name of a method, and the name of the procedure to call whenever the method is activated.

For instance, it is necessary to program a graphical VDA Tool, such as a 2D plot tool, to activate the TM_DISPLAY method whenever the Tool window is opened or redrawn. When TM_DISPLAY is triggered by such an event, a procedure is executed that contains the actual graphics commands. For a 2D plot tool, the commands might include PLOT and OPLOT. For a 3D surface tool, the SURFACE command might be called as part of this drawing procedure. It is up to the VDA Tools developer to provide the appropriate drawing procedure.

TmSetMethod, unique, ‘TM_DISPLAY’, ‘WzPlotDisplay’
TmSetMethod, unique, ‘TM_CODEGEN’, ‘WzPlotCodgen’
TmSetMethod, unique, ‘TM_RESTORE’, ‘WzPlot’
TmSetMethod, unique, ‘TM_CONVERT’, ‘WzPlotConvert’

See Also
TmEnumerateMethods, TmExecuteMethod, TmGetMethod

---

**TmStartCodeGen Procedure**

Opens a file into which PV-WAVE code is written.

**Usage**

TmStartCodeGen, filename

**Parameters**

*filename* — A string containing the name of a file in which to write the generated PV-WAVE code.

**Keywords**

None.

**Discussion**

The code generation functions allow a VDA Tool user to write the PV-WAVE code used to create a plot, import data, or any other VDA Tool action.
TmStartCodeGen opens the code generation file. Strings are written to this file with TmCodeGen. When the writing is completed, TmEndCodeGen is called to write an end statement to the file and close it.

This routines is called before the execution of the TM_CODEGEN method.

**Example**

```
TmStartCodeGen, 'AppCode.pro'
```

**See Also**

TmCodeGen, TmEndCodeGen

---

**TmUnregister Procedure**

Removes the specified VDA Tool from the Tools Manager registry.

**Usage**

```
TmUnregister, tool_name
```

**Parameters**

- `tool_name` — A string containing the unique name of a currently registered VDA Tool.

**Keywords**

None.

**Example**

The following code simply initializes the VDA Tools Manager, obtains a unique VDA Tool name, creates a top-level widget, and registers the VDA Tool. The TmEnumerateToolNames function verifies that the Tool was registered. Then, the tool is unregistered. This too is confirmed with TmEnumerateToolNames.

```
TmInit

unique = TmGetUniqueToolName('WzPlotTool')

```

```
top = WwInit('VDA', 'Examples', layout, $
```
Shell_name='WzPlotTool', Layout_name='toolArea', $Title=unique_name, /Form)
TmRegister, unique, top
PRINT, TmEnumerateToolNames()
   WzPlotTool_0
   ; Print the names of the tools currently registered with the Tools Manager.
TmUnregister, unique
PRINT, TmEnumerateToolNames()
   ; Note that the previously registered tool name is now removed.

See Also
  TmEnumerateToolNames, TmGetMessage, TmGetTop,
  TmGetUniqueToolName, TmInit, TmRegister
Graphical Elements API (Tm)

This chapter describes the new VDA Tools Graphical Element (GRAEL) API routines. These functions allow the VDA Tools developer to manipulate and add graphical elements in VDA Tools.

**TmAddGrael Procedure**

Adds a graphical element to the graphical element list for the specified instance of a VDA Tool.

**Usage**

TmAddGrael, `tool_name`, `grael_name`

**Parameters**

- `tool_name` — A string containing the unique name of a VDA Tool.
- `grael_name` — A string containing the unique name of the graphical element to add.

**Keywords**

- Rect — A two-element array specifying the upper-left and lower-right corners of a rectangle, in device coordinates. The rectangle specifies the boundary in which the user must click to select the item.
**Discussion**

This procedure allows you to add a new graphical element to a VDA Tool. To create a graphical element, you must first define it with TmSetAttribute and then associate a method with it with TmSetMethod. Use TmGetUniqueGraelName to obtain a unique name for the new graphical element.

**Example**

TmAddGrael, 'WzPlot_0', 'CIRCLE'

**See Also**

TmDelGrael, TmEnumerateGraels, TmGetGraelRectangle, TmGetUniqueGraelName, TmSetGraelRectangle

---

**TmAddSelectedGrael Procedure**

Adds a graphical element to the graphical element selection list.

**Usage**

TmAddSelectedGrael, *tool_name, grael_name*

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool.

*grael_name* — A string containing the name of the graphical element to add to the graphical element selection list.

**Keywords**

None.

**Discussion**

The graphical element selection list enables graphical elements to be cut, copied, pasted, and deleted from VDA Tools. To get a list of graphical elements associated with a given VDA Tool, use the TmEnumerateGraels function. To see which graphical elements are currently on the selection list, use TmEnumerateSelectedGraels.
If the *grael_name* is already selected (on the selection list), no action is taken.

**Example**

The following commands get the names of the items associated with a VDA Tool, then add one of the items, a graphical element (rectangle), to the selected list.

```plaintext
PRINT, TmEnumerateGraels('WzPlot_0')
TM_WINDOWID TM RECTANGLE LINE AXIS LEGEND TEXT MENUBAR
   BUTTONBAR MESSAGE XX TM_HELP AXIS_0 AXIS_1 TM_DRAWING RECTANGLE_0
   LINE_1 GROUP_0
TmAddSelectedGrael, 'WzPlot_0', 'RECTANGLE_0'
PRINT, TmEnumerateSelectedGraels('WzPlot_0')
   RECTANGLE_0
```

**See Also**

*TmDelSelectedGraels, TmEnumerateSelectedGraels*

---

**TmAxis Procedure**

Adds axes to a VDA Tool.

**Usage**

TmAxis, *tool_name*

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool.

**Keywords**

*Right* — If present and nonzero, adds an axis to the right edge of the plot data area.

*Left* — If present and nonzero, adds an axis to the left edge of the plot data area.

*Top* — If present and nonzero, adds an axis to the top edge of the plot data area.

*Bottom* — If present and nonzero, adds an axis to the bottom edge of the plot data area.

*Normal* — If present and nonzero, axes are positioned in normal coordinates.
Data — If present and nonzero, axes are positioned in data coordinates.

Thick — Specifies the thickness (in pixels) of the axes.

Color — Specifies the index of the color of the axes.

No_Draw — If present and nonzero, no axes are drawn.

Discussion

TmAxis is a graphical element (GRAEL) routine. GRAELs are predefined graphics routines used by VDA Tools. These routines allow you to add, configure, and remove graphical elements in the VDA Tool display area. The standard set of GRAELs includes axes, text, bitmaps, legends, lines, and rectangles. This standard set is accessible from the standard VDA Tool menu bar and button bar, which are provided by the VDA Utility routines WoMenuBar and WoButtonBar.

Example

The following example code is a callback routine for a menu bar. This callback executes TmAxis in response to menu selections.

```
PRO AxisCB, wid, index
  ; Go up one level in the menu.
  x = WtGet( wid, /Parent)
  tool_name = GetMenuBarToolName(x)
  ; Create the requested axis.
  CASE index OF
  1:  BEGIN            ; Left
       TmAxis, tool_name, /Left
       END
  2:  BEGIN            ; Right
       TmAxis, tool_name, /Right
       END
  3:  BEGIN            ; Top
       TmAxis, tool_name, /Top
       END
  4:  BEGIN            ; Bottom
       TmAxis, tool_name, /Bottom
       END
  ENDCASE
END
```
See Also
TmText, TmBitmap, TmLegend, TmLine, TmRect

TmBitmap Procedure

Adds a bitmap (2D array) to a VDA Tool.

Usage
TmBitmap, tool_name, bitmap_name

Parameters

*tool_name* — A string containing the unique name of a VDA Tool.

*bitmap_name* — A string containing the name of a $MAIN$ level variable containing a 2D array.

Keywords
None.

Discussion

TmBitmap is a graphical element (GRAEL) routine. GRAELs are predefined graphics routines used by VDA Tools. These routines allow you to add, configure, and remove graphical elements in the VDA Tool display area. The standard set of GRAELs includes axes, text, bitmaps, legends, lines, and rectangles. This standard set is accessible from the standard VDA Tool menu bar and button bar, which are provided by the VDA Utility routines WoMenuBar and WoButtonBar.

Example

The following line adds a bitmap called vni_logo to the VDA Tool WzPlot_0 after the user clicks MB1 in the drawing area.

TmBitmap, ‘WzPlot_0’, ‘vni_logo’

See Also
TmAxis, TmLegend, TmLine, TmRect, TmText
**TmBottomGrael Procedure**

Sets the specified graphical element to be on the bottom of the display list (displayed behind the other graphical elements).

**Usage**

TmBottomGrael, *tool_name, grael_name*

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool.

*grael_name* — A string containing the name of the graphical element to move to the bottom.

**Keywords**

None.

**Discussion**

This function is called by the Edit=>Bottom command on the graphical VDA Tools. A list of graphical elements for a given VDA Tool can be obtained with TmEnumerateGraels.

**Example**

TmBottomGrael, ‘WzPlot_0’, ‘GROUP_0’

**See Also**

TmTopGrael
TmDelGrael Procedure

Removes a specified graphical element from the list of graphical elements associated with a VDA Tool instance.

Usage
TmDelGrael, tool_name, grael_name

Parameters

tool_name — A string containing the unique name of a VDA Tool.
grael_name — A string containing the name of the graphical element to delete.

Keywords
All — When specified and nonzero, deletes all graphical elements from the specified VDA Tool. This keyword supersedes the grael_name parameter.

Discussion
To obtain a list of graphical elements associated with a given VDA Tool, use TmEnumerateGraels.

NOTE  If you specify the All keyword, you will delete all the graphical elements plus any other items that were defined for the VDA Tool. Use TmEnumerateGraels to return the list of all items to determine if the All keyword is appropriate.

Example
TmDelGrael, ‘WzPlot_0’, ‘RECTANGLE_1’

See Also
TmAddGrael, TmEnumerateGraels, TmGetGraelRectangle, TmGetUniqueGraelName, TmSetGraelRectangle
**TmDelSelectedGraels Procedure**

Deletes a graphical element from the list of selected graphical elements.

**Usage**

TmDelSelectedGraels, *tool_name*, *grael_name*

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool from which the graphical element is to be removed.

*grael_name* — A string containing the name of the graphical element to remove.

**Keywords**

*All* — When specified and nonzero, deletes all selected graphical elements from the specified VDA Tool. This keyword supersedes the *grael_name* parameter.

**Discussion**

The graphical element selection list enables graphical elements to be cut, copied, pasted, and deleted from VDA Tools. To get a list of graphical elements associated with a given VDA Tool, use the TmEnumerateGraels function. To see which graphical elements are currently on the selection list, use TmEnumerateSelectedGraels.

**NOTE** If you specify the *All* keyword, you will delete all the graphical elements plus any other items that were defined for the VDA Tool. Use TmEnumerateSelectedGraels to return the list of all items to determine if the *All* keyword is appropriate.

**Example**

The first command returns all of the graphical elements currently on the selection list for the VDA Tool WzPlot_0. Then, one of the elements, RECTANGLE_0 is deleted from the list.

```plaintext
PRINT, TmEnumerateSelectedGraels(‘WzPlot_0’)
  RECTANGLE_0 LINE_0 LINE_1
TmDelSelectedGraels, ‘WzPlot_0’, ‘RECTANGLE_0’
```
PRINT, TmEnumerateSelectedGraels('WzPlot_0')
LINE_0 LINE_1

See Also
TmAddSelectedGrael, TmEnumerateSelectedGraels

---

**TmEnumerateGraelMethods Function**

Obtain a list of all the methods set for a graphical element in a specified VDA Tool.

**Usage**

\[
\text{methods} = \text{TmEnumerateGraelMethods} (\text{tool\_name}, \text{grael\_name})
\]

**Parameters**

- `tool_name` — A string containing the unique name of a VDA Tool.
- `grael_name` — A string containing the name of a graphical element associated with the VDA Tool.

**Returned Value**

- `methods` — A string array containing the names of all the methods registered for a graphical element in the specified VDA Tool.

**Keywords**

None.

**Example**

This command lists all the methods associated with an axis. By convention, methods always begin with “TM_”.

PRINT, TmEnumerateGraelMethods('WzPlot_0', 'AXIS_0')
   TM_HIGHLIGHT TM_DIALOG TM_CODEGEN TM_STARTMOVE TM_CONVERT
   TM_DISPLAY TM_SELECTGRAEL TM_UNHIGHLIGHT TM_RELOCATE

See Also
TmExecuteGraelMethod, TmGetGraelMethod, TmSetGraelMethod
**TmEnumerateGraels Function**

Returns all the graphical elements that were set for a given VDA Tool.

**Usage**

\[
grael\_list = TmEnumerateGraels(tool\_name)\]

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool.

**Returned Value**

*grael_list* — An array of strings containing the graphical elements that were set for the given VDA Tool.

**Keywords**

None.

**Discussion**

Before you can manipulate a graphical element, you must know its name. Use this function to return the names of graphical elements in a VDA Tool. With the name, you can use functions like TmGetAttribute, TmSetAttribute, TmSetGraelMethod, and other routines that perform graphical element operations.

**Example**

List the graels currently registered with the given VDA Tool.

```plaintext
PRINT, TmEnumerateGraels('WzPlot_0')
```

**See Also**

TmAddGrael, TmDelGrael, TmGetGraelRectangle, TmGetUniqueGraelName, TmSetGraelRectangle
**TmEnumerateSelectedGraels Function**

Obtains a list of graphical elements or other items currently on the graphical items selection list.

**Usage**

\[ graels = TmEnumerateSelectedGraels(tool\_name) \]

**Parameters**

*tool\_name* — A string containing the unique name of a VDA Tool.

**Returned Value**

*graels* — A string array containing all the names of the graphical elements or other items on the selection list for the VDA Tool.

**Keywords**

None.

**Discussion**

The graphical element selection list enables graphical elements to be cut, copied, pasted, and deleted from VDA Tools. To get a list of graphical elements associated with a given VDA Tool, use the TmEnumerateGraels function.

**Example**

The following commands add graphical elements to the selection list and then print out the contents of the list.

```plaintext
PRINT, TmEnumerateGraels('WzPlot_0')
   TM_WINDOWID TM RECTANGLE LINE AXIS LEGEND TEXT MENUBAR
   BUTTONBAR MESSAGE XX TM_HELP AXIS_0 AXIS_1 TM_DRAWING RECTANGLE_0
   LINE_1 GROUP_0
TmAddSelectedGrael, 'WzPlot_0', 'RECTANGLE_0'
PRINT, TmEnumerateSelectedGraels('WzPlot_0')
   RECTANGLE_0
```

**Conclusion**

The TmEnumerateSelectedGraels function is a powerful tool for interacting with the graphical elements within VDA Tools, allowing for efficient management and manipulation of these elements. Its flexibility and ease of use make it a valuable asset in developing VDA tool applications.
TmExecuteGraelMethod Procedure

Executes a method for a graphical method based on the method name.

Usage

TmExecuteGraelMethod, tool_name, grael_name, method_name

Parameters

tool_name — A string containing the unique name of a VDA Tool.
grael_name — A string containing the name of the graphical element.
method_name — A string containing the name of the method to execute. e.g.,
TM_DISPLAY.

Keywords

None.

Discussion

If the named method was not previously registered with the Tools Manager, no
action is taken.

Example

TmSetGraelMethod, ‘WzPlot_0’, ‘LINE_0’, ‘TM_DISPLAY’, $
‘TmLine’
TmExecuteGraelMethod, ‘WzPlot_0’, ‘LINE_0’, ‘TM_DISPLAY’

See Also

TmEnumerateGraelMethods, TmGetGraelMethod, TmSetGraelMethod
**TmGetGraelMethod Function**

Obtains the data structure for the specified method.

**Usage**

\[ name = \text{TmGetGraelMethod}(\text{tool\_name}, \text{grael\_name}, \text{method\_name}) \]

**Parameters**

- **tool\_name** — A string containing the unique name of a VDA Tool.
- **grael\_name** — A string containing the name of the graphical element.
- **method\_name** — A string containing the name of the method to execute, e.g., TMDISPLAY.

**Returned Value**

\[ name \] — The data structure of the method.

**Keywords**

None.

**Discussion**

The method data structure contains two tags: the name of the method procedure and a variable for user data.

**Example**

This command returns the name of the procedure that is called (TmLine) when the TMDISPLAY method is called for the graphical element LINE_0.

\[
\text{INFO, /Full, TmGetGraelMethod(‘WzPlot_0’, ‘LINE_0’, ‘TM_DISPLAY’) }
\{ \text{TmLine{}} \}
\]

**See Also**

TmEnumerateGraelMethods, TmExecuteGraelMethod, TmSetGraelMethod
**TmGetGraelRectangle Function**

Returns the rectangular boundary of a graphical element.

**Usage**

\[ rect = \text{TmGetGraelRectangle}(\text{tool\_name}, \text{grael\_name}) \]

**Parameters**

- **tool\_name** — A string containing the unique name of a VDA Tool.
- **grael\_name** — A string containing the name of the graphical element.

**Returned Value**

- **rect** — A four-element array defining the endpoints of a rectangle in device coordinates: \([x_1 y_1 x_2 y_2]\).

**Keywords**

None.

**Discussion**

The rectangular boundary is a region in which the user can click to select the graphical element. This rectangle is also the highlighted border that indicates when a graphical element has been selected.

**Example**

First, the graphical elements for a VDA Tool are enumerated. Then, the coordinates of the bounding rectangle for one of the graphical elements is returned by \text{TmGetGraelRectangle}.

```
PRINT, TmEnumerateGraels('WzPlot_0')
  TM_WINDOWID TM RECTANGLE LINE AXIS LEGEND TEXT MENUBAR
  BUTTONBAR MESSAGE XX TM_HELP AXIS_0 AXIS_1 TM_DRAWING
PRINT, TmGetGraelRectangle('WzPlot_0', 'AXIS_0')
  50.0025  38.0034  492.003  58.0034
```
TmGetUniqueGraelName Function

Obtains a unique name based on the name of the specified graphical element.

Usage

\[
name = \text{TmGetUniqueGraelName}(\text{tool\_name}, \text{grael\_name})
\]

Parameters

- **tool\_name** — A string containing the unique name of a VDA Tool.
- **grael\_name** — A string containing the name of the graphical element.

Returned Value

- **name** — A string containing a unique name for the graphical element.

Keywords

None.

Discussion

Use this function to get a unique graphical element name before using the TmAddGrael function.

The unique name returned by this function is a variation on the name provided by the **grael\_name** input parameter (i.e., LINE\_2, if LINE\_0 and LINE\_1 already exist).

Example

\[
\text{PRINT, TmGetUniqueGraelName('WzPlot\_0', 'LINE')}
\]

  
  LINE\_0


**See Also**

TmAddGrael, TmDelGrael, TmEnumerateGraels, TmGetGraelRectangle, TmSetGraelRectangle

---

**TmGroupGraels Function**

Groups a number of selected graphical elements as one graphical element with a unique name.

**Usage**

\[ name = \text{TmGroupGraels}(\text{tool\_name}, \text{grael\_names}) \]

**Parameters**

- **tool\_name** — A string containing the unique name of a VDA Tool.
- **grael\_names** — An array of strings containing graphical element names for graphical elements associated with the specified VDA Tool.

**Returned Value**

- **name** — A string containing a unique name for the group of graphical elements, e.g., GROUP_01.

**Keywords**

None.

**Discussion**

This function is called when the Edit=>Group command is called from the graphical menu.

**Example**

The first command lists all the graphical elements in a VDA Tool. The TmGroupGraels function is used to group several of the graphical elements — two rectangles and two lines — that have been added to the plot window. The function
returns a unique name for the group. In the plot window, the grouped items are surrounded by a highlighted border.

```plaintext
PRINT, TmEnumerateGraels('WzPlot_0')
   TM_WINDOWID TM RECTANGLE LINE AXIS LEGEND TEXT MENUBAR
   BUTTONBAR MESSAGE XX TM_HELP AXIS_0 AXIS_1 TM_DRAWING RECTANGLE_0
   RECTANGLE_1 RECTANGLE_2 LINE_0 LINE_1 LINE_2
grp = TmGroupGraels('WzPlot_0', ['RECTANGLE_0', '
   'RECTANGLE_1', 'LINE_0', 'LINE_2'])
PRINT, grp
   GROUP_0

See Also
TmUngroupGraels
```

---

**TmLegend Procedure**

Adds a legend to a VDA Tool. The exact size and position of the legend is determined interactively by the user.

**Usage**

```plaintext
TmLegend, tool_name
```

**Parameters**

- **tool_name** — A string containing the unique name of a VDA Tool.

**Keywords**

None.

**Discussion**

TmLegend is a graphical element (GRAEL) routine. GRAELs are predefined graphics routines used by VDA Tools. These routines allow you to add, configure, and remove graphical elements in the VDA Tool display area. The standard set of GRAELs includes axes, text, bitmaps, legends, lines, and rectangles. This standard set is accessible from the standard VDA Tool menu bar and button bar, which are provided by the VDA Utility routines WoMenuBar and WoButtonBar.
To draw a legend, the user presses and drags MB1 to define the legend border.

**Example**

The following example code is a callback routine for a menu bar. This callback executes `TmLegend` in response to a menu selection.

```pro
PRO CreateLegendCB, wid, index
    tool_name = GetMenuBarToolName(wid)
    toggle_buttons = ['dataselect', 'graelselect', 'text', 'line', 'rectangle', 'legend']
    buttons_up = [0, 0, 0, 0, 0, 0]
    WoButtonBarSet, tool_name, toggle_buttons, buttons_up
    WoButtonBarSet, tool_name, 'legend', 1
    WoAddMessage, tool_name, 'MSG_LegendCreate', /Clear
    WoAddStatus, tool_name, 'MSG_LegendCreateStatus', Item='STATUS2'
    TmLegend, tool_name
END
```

**See Also**

`TmAxis`, `TmBitmap`, `TmLine`, `TmRect`, `TmText`

---

**TmLine Procedure**

Adds a line to a VDA Tool. The exact length and position of the line is determined interactively by the user.

**Usage**

`TmLine, tool_name`

**Parameters**

`tool_name` — A string containing the unique name of a VDA Tool.

**Keywords**

None.
Discussion

TmLine is a graphical element (GRAEL) routine. GRAELs are predefined graphics routines used by VDA Tools. These routines allow you to add, configure, and remove graphical elements in the VDA Tool display area. The standard set of GRAELs includes axes, text, bitmaps, legends, lines, and rectangles. This standard set is accessible from the standard VDA Tool menu bar and button bar, which are provided by the VDA Utility routines WoMenuBar and WoButtonBar.

To create a line, the user presses and drags MB1 to define the endpoints.

Example

The following example code is a callback routine for a menu bar. This callback executes TmLine in response to a menu selection.

```plaintext
PRO CreateLineCB, wid, index
    tool_name = GetMenuBarToolName(wid)
    toggle_buttons = ['dataselect', 'graelselect', 'text', $
                      'line', 'rectangle', 'legend']
    buttons_up = [0, 0, 0, 0, 0, 0]
    WoButtonBarSet, tool_name, toggle_buttons, buttons_up
    WoButtonBarSet, tool_name, 'line', 1
    WoAddMessage, tool_name, 'MSG_LineCreate', /Clear
    WoAddStatus, tool_name, 'MSG_LineCreateStatus', Item='STATUS2'
    TmLine, tool_name
END
```

See Also

TmAxis, TmBitmap, TmLegend, TmRect, TmText
**TmRect Procedure**

Adds a rectangle to a VDA Tool. The exact size and position of the rectangle is determined interactively by the user.

**Usage**

TmRect, tool_name

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool.

**Keywords**

None.

**Discussion**

TmRect is a graphical element (GRAEL) routine. GRAELs are predefined graphics routines used by VDA Tools. These routines allow you to add, configure, and remove graphical elements in the VDA Tool display area. The standard set of GRAELs includes axes, text, bitmaps, legends, lines, and rectangles. This standard set is accessible from the standard VDA Tool menu bar and button bar, which are provided by the VDA Utility routines *WoMenuBar* and *WoButtonBar*.

To create a rectangle, the user presses and drags MB1 to define the corners of the rectangle.

**Example**

The following example code is a callback routine for a menu bar. This callback executes TmRect in response to a menu selection.

```pro
PRO CreateBoxCB, wid, index
    tool_name = GetMenuBarToolName(wid)
    toggle_buttons = ['dataselect', 'graelselect', 'text', $ 'line', 'rectangle', 'legend']
    buttons_up = [0, 0, 0, 0, 0, 0]
    WoButtonBarSet, tool_name, toggle_buttons, buttons_up
    WoButtonBarSet, tool_name, 'rectangle', 1
```
WoAddMessage, tool_name, 'MSG_RectCreate', /Clear
WoAddStatus, tool_name, 'MSG_RectCreateStatus', $ Item='STATUS2'
  TmRect, tool_name
END

See Also

TmAxis, TmBitmap, TmLegend, TmLine, TmText

---

**TmSetGraelMethod Procedure**

Sets the name of the method procedure for a given method name and graphical element.

**Usage**

TmSetGraelMethod, *tool_name*, *grael_name*, *method_name*, *method_value*

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool.

*grael_name* — A string containing the name of the graphical element.

*method_name* — A string containing the name of the method to set for the specified graphical element (i.e., TM_DISPLAY).

*value* — A string containing the name of the method procedure (for example, ‘DrawAxis’).

**Keywords**

*user_data* — A string containing data associated with the method.

**Example**

TmAddGrael, ‘WzPlot_0’, ‘CIRCLE’
; Add a new graphical element called "CIRCLE" to a VDA Tool.
TmSetGraelMethod, 'WzPlot_0', 'CIRCLE', 'TMDISPLAY', 'DrawCircle'
; Set a method for the new graphical element. This method calls a procedure,
; DrawCircle, which performs the graphics operations to draw circles in the plot
; window.

See Also
TmEnumerateGraelMethods, TmExecuteGraelMethod, TmGetGraelMethod

---

**TmSetGraelRectangle Procedure**

Sets the selection rectangle for a graphical element, or a set of graphical elements.

**Usage**

TmSetGraelRectangle, *tool_name*, *grael_name*, *rectangle*

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool.
*grael_name* — A string containing the name of the graphical element.
*rectangle* — A four-element array defining the endpoints of a rectangle in device coordinates: \([x_1 \, y_1 \, x_2 \, y_2]\).

**Keywords**

None.

**Discussion**

The rectangular boundary is a region in which the user can click to select the graphical element. This rectangle is also the highlighted border that indicates when a graphical element has been selected.

Use this function to reset the size of a graphical element’s boundary rectangle after the graphical element has been resized or moved.

**Example**

The following commands reset the bounding box for an axis.
rect = [50, 227, 492, 247]
TmSetGraelRectangle, ‘WzPlot_0’, ‘AXIS_0’, rect

See Also
TmAddGrael, TmDelGrael, TmEnumerateGraels, TmGetGraelRectangle, TmGetUniqueGraelName

---

**TmText Procedure**

Adds text to a VDA Tool. The position of the text and the text itself are determined interactively by the user.

**Usage**

TmText, *tool_name*

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool.

**Keywords**

None.

**Discussion**

TmText is a graphical element (GRAEL) routine. GRAELs are predefined graphics routines used by VDA Tools. These routines allow you to add, configure, and remove graphical elements in the VDA Tool display area. The standard set of GRAELs includes axes, text, bitmaps, legends, lines, and rectangles. This standard set is accessible from the standard VDA Tool menu bar and button bar, which are provided by the VDA Utility routines WoMenuBar and WoButtonBar.

To add text, the user clicks MB1 where the text is to start, enters the text, and presses <Return> when finished.
Embedded Functions in Text Graels

You can embed PV-WAVE functions inside a string used in a text grael (a graphical element that includes the VDA Tools Manager function TmText). This feature is useful for embedding a calculated value to a string. When the string is displayed, the function is executed and the result is inserted into the string.

The format for embedded functions is:

```
%%FUNCTION(args):(FORMAT)%%
```

where:

**FUNCTION** — Any valid PV-WAVE function that returns a scalar value. All variables used by the function, including variables used as values for keywords, must exist at the $MAIN$ level of PV-WAVE.

**FORMAT** — A valid FORTRAN-style format. For detailed information on output formats, see Appendix A, *FORTRAN and C Format Strings*, in the *PV-WAVE Programmer’s Guide*.

Example 1

The following example code is a callback routine for a menu bar. This callback executes TmText in response to a menu selection.

```
PRO CreateTextCB, wid, index
    tool_name = GetMenuBarToolName(wid)
    toggle_buttons = ['dataselect', 'graelselect', 'text', '$line', 'rectangle', 'legend']
    buttons_up = [0, 0, 0, 0, 0, 0]
    WoButtonBarSet, tool_name, toggle_buttons, buttons_up
    WoButtonBarSet, tool_name, 'text', 1
    WoAddMessage, tool_name, 'MSG_TextCreate', /Clear
    WoAddStatus, tool_name, 'MSG_TextCreateStatus', Item='STATUS2'
    TmText, tool_name
END
```

Example 2

Assume that the variable *my_var* exists at $MAIN$. When the following string is displayed in a text grael, the value of the AVG function is calculated and displayed in the specified format.
The average of my_var is: \%AVG(my_var):(F8.3)\%

You can use additional arguments and keywords, as long as all named parameters exist at $MAIN$. You can also use expressions as parameters, as follows:

The median of DIST(200) is: \%MEDIAN(DIST(200))\%

See Also

TmAxis, TmBitmap, TmLegend, TmLine, TmRect

---

**TmTopGrael Procedure**

Sets the specified graphical element to be at the top of the display list (displayed in front of other graphical elements).

**Usage**

TmTopGrael, tool_name, grael_name

**Parameters**

*tool_name* — A string containing the unique name of a VDA Tool.

*grael_name* — A string containing the name of the graphical element to move to the top.

**Keywords**

None.

**Discussion**

This function is called by the Edit\=>Top command on the graphical VDA Tools. A list of graphical elements for a given VDA Tool can be obtained with TmEnumerateGraels.

**Example**

TmTopGrael, ‘WzPlot_0’, ‘RECTANGLE_1’
See Also
TmBottomGrael

TmUngroupGraels Procedure

Ungroups a group of graphical elements.

Usage
TmUngroupGraels, tool_name, group_name

Parameters

tool_name — A string containing the unique name of a VDA Tool.

group_name — A string containing the name of a graphical element group.

Keywords
None.

Discussion
The group_name parameter is a string returned by the TmGroupGraels function.

Example
This example simply groups and then ungroups some graphical elements.

grp = TmGroupGraels('WzPlot_0', ['RECTANGLE_0', $
  'RECTANGLE_1', 'LINE_0', 'LINE_2'])
PRINT, grp
  GROUP_0
TmUngroupGraels, 'WzPlot_0', 'GROUP_0'

See Also
TmGroupGraels
VDA Tools Utilities (Wo)

This chapter describes the VDA Tools Utilities (Wo) routines. This set of convenience routines help you to develop a VDA Tool user interface quickly and efficiently.

**WoAddButtons Procedure**

Adds a bank of buttons to a button bar.

**Usage**

WoAddButtons, *toolname, buttons*

**Input Parameters**

*toolname* — (string) Specifies the unique name of the VDA Tool to which the button bar is attached.

*buttons* — An unnamed structure containing the button definitions.

**Keywords**

*Measure* — Specifies the number of columns of buttons (for a vertical box) or rows (for a horizontal box).

*Radio* — If nonzero, the buttons have a “one of many” behavior.
**Sensitive** — Specifies an array of initial sensitivity settings for the buttons. A value of 0 makes the button insensitive; 1 makes the button sensitive. (Default: All buttons are sensitive.)

**Vertical** — When nonzero, creates a vertically aligned column of buttons.

**Discussion**

An unnamed structure has the following general definition:

\[ x = \{ \text{tag_name}_1: \text{tag_def}_1, \text{tag_name}_n: \text{tag_def}_n \} \]

The following tag names and tag definitions can be used in the unnamed structure used to define buttons:

- **LAYOUT_NAME**: 'name' — Specifies the name of the row/column layout used to organize the buttons.
- **DESCRIPTOR**: 'name' — Specifies the descriptor name for the button. A descriptor is a string used to identify a button. This string is also used as the button’s widget ID.
- **CALLBACK**: 'name' — Specifies the name of the callback routine for the button.
- **STATUS_CALLBACK**: 'name' — Specifies the name of the routine that prints the name of the button when the pointer passes over the button.
- **USERDATA**: variable — Specifies a variable to pass to the callback when the button is pressed.
- **SENSITIVE_PIXMAP**: 'pathname' — Specifies the full pathname of the insensitive pixmap for the button.
- **PIXMAP**: 'pathname' — Specifies the full pathname of the sensitive pixmap for the button.

For an example of an unnamed structure of button definitions, see the file wographicsbuttons.pro in:

**(UNIX)** <wavedir>/lib/vdatools

**(OpenVMS)** <wavedir>:\LIB.VDATOOLS

**(Windows)** <wavedir>\lib\vdatools

Where <wavedir> is the main PV=WAVE directory.
Example

The following code shows how to add custom buttons to the standard button bar. First, button callbacks are defined, and then a structure is created to define two buttons. Finally, WoAddButtonBar is used to add the buttons to the standard button bar. WoButtonBarSetSensitivity is used to set the sensitivity of one of the buttons, and WoButtonBarSet is used to press one of the buttons.

```plaintext
PRO Button0StatusCB, wid, tool_name, event
    WoAddStatus, tool_name, 'MSG_Button0Status'
END
PRO Button1StatusCB, wid, tool_name, event
    WoAddStatus, tool_name, 'MSG_Button0Status'
END
PRO Button0CB, wid, tool_name
    print, 'Button 0 pressed in tool ' + tool_name
    WoAddMessage, tool_name, 'MyTool_Button0', /Clear
END
PRO Button1CB, wid, tool_name
    print, 'Button 1 pressed in tool ' + tool_name
    WoAddMessage, tool_name, 'MyTool_Button0', /Clear
END

; Define the button bar structure.

pixmap_directory='~user/Pixmaps/
MyButtons = $
{, $
    LAYOUT_NAME: 'MyButtons', $
    DESCRIBER: 'MyButton_0', $
    CALLBACK: 'Button0CB', $
    STATUS_CALLBACK: 'Button0StatusCB', $
    INSENSITIVE_PIXMAP: pixmap_dir + 'button0x.pm', $
    PIXMAP: pixmap_dir + 'button0.pm', $
    DESCRIBER: 'MyButton_1', $
    CALLBACK: 'Button1CB', $
    STATUS_CALLBACK: 'Button1StatusCB', $
```
INSENSITIVE_PIXMAP: pixmap_dir + 'button1x.pm', 
PIXMAP: pixmap_dir + 'button1.pm' 
}

; Create the standard graphics button bar.
tb = WoButtonBar(layout, tool_name, Top=bar, /Graphics, 
/Left, /Right)

; Add the additional buttons.
WoAddButtons, tool_name, MyButtons

; Make the second button insensitive.
WoButtonBarSetSensitivity, tool_name, 'MyButton_1', 0

; Push the first button.
WoButtonBarSet, tool_name, 'MyButton_0', 1

... 

See Also
WoButtonBar, WoButtonBarSet, WoButtonBarSetSensitivity

---

**WoAddMessage Procedure**

Adds a message to a message area created by WoMessage.

**Usage**

WoAddMessage, *toolname, message_key*

**Input Parameters**

*toolname* — (string) Specifies the unique name of the VDA Tool to which the message area is attached.

*message_key* — A string used to identify a message in a string resource file.

**Keywords**

*Clear* — If nonzero, clears the message area before displaying the message. (Default: Append the message after the currently displayed messages.)

*Message* — Specifies a string containing a message to display in the message area.
Discussion

The WoBuildResourceFilename function is used to return the directory in which string resources are stored.

NOTE We recommend that you use the Message keyword only if absolutely necessary. In general, “hard coding” messages with this keyword impedes your ability to customize or internationalize your application.

Example

For an example that uses WoAddMessage, see WoMessage.

See Also

WoMessage

WoAddStatus Procedure

Display a message in the status bar of a VDA Tool.

Usage

WoAddStatus, toolname, status_key

Input Parameters

toolname — (string) Specifies the unique name of the VDA Tool to which the status bar is attached.

status_key — A string used to identify a message in a string resource file.

Keywords

Status — Specifies a string containing a status message to display in the status bar.

Discussion

The WoBuildResourceFilename function is used to return the directory in which string resources are stored.
NOTE  We recommend that you use the Status keyword only if absolutely necessary. In general, “hardcoding” messages with this keyword impedes your ability to internationalize your application.

Example

For an example that uses WoAddStatus, see WoStatus.

See Also

WoStatus,  WoMessage

**WoBuildResourceFilename Function**

Returns the full path name for a specified resource file.

**Usage**

\[
\text{resource} \_\text{file} = \text{WoBuildResourceFilename} (\text{file})
\]

**Returned Value**

*file* — The name of the resource file.

**Keywords**

*Appdir* — A string that specifies the application directory name. This is the directory in which the application searches for resource files, string resource files, and icon files. See the **Discussion**. (Default: vdatools)

*Subdir* — A string specifying a resource file subdirectory. See the **Discussion**.

**Discussion**

**UNIX USERS**  By default, the function looks for *file* first in directories specified by the environment variable WAVE RESPATH. This environment variable is a colon separated list of directories, similar to the PV-WAVE WAVE PATH environment variable. If not found in a WAVE RESPATH directory, the directory $WAVE\_DIR/xres/!Lang/vdatools is searched, where !Lang represents the value of the !Lang system variable in PV-WAVE.
**OpenVMS USERS** By default, the function looks for file first in directories specified by the logical `WAVE_RESPATH`. This logical is a comma separated list of directories and text libraries, similar to the OpenVMS `WAVE_PATH` logical. If not found in a `WAVE_RESPATH` directory, the directory `WAVE_DIR:[XRES.!Lang.VDATOOLS]` is searched, where `!Lang` represents the value of the `!Lang` system variable in PV-WAVE.

**Windows USERS** By default, the function looks for file first in directories specified by the environment variable `WAVE_RESPATH`. This environment variable is a semicolon separated list of directories, similar to the PV-WAVE `WAVE_PATH` environment variable. If not found in a `WAVE_RESPATH` directory, the directory `C:\WAVE_DIR\xres\!Lang\vdatools` is searched, where `!Lang` represents the value of the `!Lang` system variable in PV-WAVE.

If `Subdir` is specified, the file is searched for in:

- (UNIX) `<wavedir>/xres/subdir/vdatools`
- (OpenVMS) `<wavedir>:[XRES.SUBDIR.VDATOOLS]`
- (Windows) `<wavedir>\xres\subdir\vdatools`

Where `<wavedir>` is the main PV-WAVE directory.

If `Appdir` is specified, the application searches for resources in the following directory:

- (UNIX) `<wavedir>/xres/!Lang/appdir`
- (OpenVMS) `<wavedir>:[XRES.!Lang.APPDIR]`
- (Windows) `<wavedir>\xres\!Lang\appdir`

Where `<wavedir>` is the main PV-WAVE directory.

If `Subdir` and `Appdir` are specified, the application searches for resources in the following directory:

- (UNIX) `<wavedir>/xres/subdir/appdir`
- (OpenVMS) `<wavedir>:[XRES.SUBDIR.APPDIR]`
- (Windows) `<wavedir>\xres\subdir\appdir`

Where `<wavedir>` is the main PV-WAVE directory.

If the file is not already in the resource database, the full path name is returned.
Example

The following commands are taken from the code for the WzContour VDA Tool. The full path name of the resource file for WzContour is returned and is passed to the Resource keyword of WwInit.

```
resource_file = WoBuildResourceFilename('wzcontour.ad')
top = WwInit('WzContour', 'VDATools', layout, 'WoDestroyCB', $
  Shell_name='WzContour', Layout_name='toolArea', $
  Title=unique_name, /Form, ConfirmClose='WoConfirmClose', $ 
  Resource=resource_file, Userdata=unique_name)
```

See Also

WoLoadResources, WoLoadStrings, WwInit

For information on environment variables and logicals used with PV-WAVE, see the PV-WAVE Programmer’s Guide.

---

**WoButtonBar Function**

Creates a predefined, two-row button bar that can be included in a VDA Tool.

**Usage**

```
bb_parent = WoButtonBar(parent, toolname, [buttons])
```

**Input Parameters**

*parent* — Specifies the widget ID of the parent of the button bar (long). This is the ID of the row/column container that holds the row/column container with the drawn buttons. (See the Discussion.)

*toolname* — (string) Specifies the unique name of the VDA Tool to which the button bar is to be attached.

*buttons* — (optional) Specifies an unnamed structure containing the button definitions. This parameter is not needed if the Graphics keyword is specified. See the Discussion for more information.
Returned Value

*bb_parent* — The parent widget ID of the bank of buttons (long). This is the ID of the row/column container that holds the drawn buttons. (See the *Discussion*.)

Keywords

*Graphics* — If nonzero, places a set of predefined buttons in the VDA Tool. See the *Discussion* section for more information.

*Measure* — Specifies the number of columns of buttons (for a vertical box) or rows (for a horizontal box).

*Position* — A two-element array specifying the x, y coordinates of the buttonbar inside the bulletin board widget. (Default: [0,0])

*Radio* — If nonzero, causes the buttons to behave like radio buttons, where only one button can be selected at a time.

*Sensitive* — A scalar or array specifying the initial sensitivity of the button or buttons. A value of 0 makes the button insensitive; 1 makes the button sensitive. Use *WoButtonBarSetSensitivity* to change the sensitivity after the buttons are created.

*Spacing* — Specifies the amount of space in pixels between buttons. The default is 0.

*Vertical* — When nonzero, creates a vertically aligned column of buttons.

Attachment Keywords

*Bottom* — If a widget ID is specified (for example, *Bottom=wid*), then the bottom of the button bar is attached to the top of the specified widget. If no widget ID is specified (for example, */Bottom*), then the bottom of the button bar is attached to the bottom of the parent widget.

*Left* — If a widget ID is specified (for example, *Left=wid*), then the left side of the button bar is attached to the right side of the specified widget. If no widget ID is specified (for example, */Left*), then the left side of the button bar is attached to the left side of the parent widget.

*Right* — If a widget ID is specified (for example, *Right=wid*), then the right side of the button bar is attached to the left side of the specified widget. If no widget ID is specified (for example, */Right*), then the right side of the button bar is attached to the right side of the parent widget.

*Top* — If a widget ID is specified (for example, *Top=wid*), then the top of the button bar is attached to the bottom of the specified widget. If no widget ID is specified
(for example, /Top), then the top of the button bar is attached to the top of the parent widget.

**Discussion**

The standard button bar is created as a set of drawn buttons inside a row/column container, which is placed inside another row/column container widget. The use of two row/column containers allows a standard button bar to be created that has multiple banks of buttons where each bank can have unique spacing and behavior (e.g., radio buttons).

**Using the Standard Button Bar**

If the *Graphics* keyword is specified, a predefined set of standard buttons is provided automatically. These predefined buttons are equipped with functional callbacks. If you choose to use the predefined set of buttons in your application, you do not need to modify the underlying structure of the buttons or the callbacks. Both the predefined callbacks and the underlying button structures are defined in files in the *vdatools* subdirectory of the Standard Library:

- `wographicsbuttons.pro` — Uses unnamed structures to define pixmaps and callbacks for the buttons.
- `wographicsbuttonscb.pro` — Contains the callback routines for each button.

**Using Your Own Button Bar**

If you do not choose to use the standard button bar, then you can use these files as templates for creating a customized button bar. Use the optional *button* parameter to specify the unnamed structure defining the pixmaps and callbacks for the custom button bar. For a customized button bar, you will also have to write and include appropriate callback procedures in your application.

An unnamed structure has the following general definition:

\[
\{, \text{tag}_1: \text{def}_1, \text{tag}_n: \text{def}_n\}
\]

The following tag names and tag definitions can be used in the unnamed structure used to define buttons:

- **LAYOUT_NAME**: `'name'` — Specifies the name of the row/column layout used to organize the buttons.
• DESCRIPTOR: ‘name’ — Specifies the a descriptor name for the button. A
descriptor is a string used to identify a button. This string is also used as the
button’s widget ID.

• CALLBACK: ‘name’ — Specifies the name of the callback routine for the
button.

• STATUS_CALLBACK: ‘name’ — Specifies the name of the routine that
prints the name of the button when the pointer passes over the button.

• USERDATA: variable — Specifies a variable to pass to the callback when the
button is pressed.

• INSENSITIVE_PIXMAP: ‘pathname’ — Specifies the full pathname of the
insensitive pixmap for the button.

• PIXMAP: ‘pathname’ — Specifies the full pathname of the sensitive pixmap
for the button.

For an example of an unnamed structure of button definitions, see the file
wographicsbuttons.pro in:

(UNIX)   <wavedir>/lib/vdatools
(OpenVMS) <wavedir>:[LIB.VDATOOLS]
(Windows) <wavedir>\lib\vdatools

Where <wavedir> is the main PV-WAVE directory.

NOTE For information on how to use the functions on the standard button bar, run
one of the graphical VDA Tools, such as WzPlot, and read about the button bar in
online Help.

Example

This call to WoButtonBar is taken from the VDA Tool template file:

(UNIX)   <wavedir>/lib/vdatools/wztemplate.pro
(OpenVMS) <wavedir>:[LIB.VDATOOLS]WZTEMPLATE.PRO
(Windows) <wavedir>\lib\vdatools\wztemplate.pro

Where <wavedir> is the main PV-WAVE directory.

tb = WoButtonBar(layout, unique_name, Top=bar, /Graphics, $
    /Left, /Right)
    ; Creates a standard button bar for a VDA Tool.
WoButtonBarSet Procedure

Changes the setting of a button in a button bar.

Usage

WoButtonBarSet, toolname, descriptor, setting

Input Parameters

toolname — (string) Specifies the unique name of the VDA Tool to which the button bar is attached.

descriptor — Specifies a scalar string or string array containing the descriptor of the button or buttons to set.

setting — Specifies a scalar or array with the new setting or settings. A value of 0 unsets the button; 1 sets the button.

Keywords

None.

Discussion

If the setting parameter is a scalar, then all buttons are set to the value of setting. If the setting parameter is an array, each button is set to its corresponding element in setting. In other words, the first button is set to the first value in the array, the second button to the second value, and so on. If setting is an array with fewer elements than buttons, the extra buttons are set to 1.

A descriptor is a string used to identify a button. This string is also used as the button’s widget name. The descriptor is set in the standard button bar structure. The standard graphics button bar is defined in the file wographicsbuttons.pro in:

(UNIX)  <wavedir>/lib/vdatools
(OpenVMS)  <wavedir>:/LIB.VDATOOLS]
WoButtonBarSetSensitivity Procedure

Sets the sensitivity of one or more buttons on a button bar.

Usage

WoButtonBarSetSensitivity, toolname, descriptor, sensitivity

Input Parameters

toolname — (string) Specifies the unique name of the VDA Tool to which the button bar is attached.

descriptor — Specifies a scalar string or string array containing the descriptor of the button or buttons to set.

sensitivity — Specifies a scalar or array with the sensitivity of the button or buttons. A value of 0 makes the button insensitive; 1 makes the button sensitive.

Keywords

None.

Discussion

If the sensitivity parameter is a scalar, then all buttons are set to the value of sensitivity. If the sensitivity parameter is an array, each button is set to its corresponding element in sensitivity. In other words, the sensitivity of the first button is set to the first value in the array, the second button to the second value, and so on. If sensitivity...
ity is an array with fewer elements than buttons, the extra buttons are set to 1 (insensitive).

A descriptor is a string used to identify a button. This string is also used as the button’s widget ID. The descriptor is set in the standard button bar structure. The standard graphics button bar is defined in the file wographicsbuttons.pro in:

(UNIX)  <wavedir>/lib/vdatools
(OpenVMS)  <wavedir>:[LIB.VDATOOLS]
(Windows)  <wavedir>\lib\vdatools

Where <wavedir> is the main PV-WAVE directory.

Example

For an example that uses WoButtonBarSetSensitivity, see WoAddButtons.

See Also

WoAddButtons, WoButtonBar, WoButtonBarSet

---

**WoCheckFile Function**

Confirms if a file is readable or writable.

**Usage**

\[status = WoCheckFile(file)\]

**Input Parameters**

*file* — A string containing the name of the file to check.

**Returned Value**

*status* — If 1, the file can be used for the given operation; if 0, the file cannot be used.
Input Keywords

Read — If specified and nonzero, the function verifies that the file is readable.
Write — If specified and nonzero, the function verifies that the file is writable.

Output Keyword

FullName — Returns a string containing the expanded filename. Constructs like ~user and $ENV_VAR are expanded.
Size — Returns the size of the file in bytes.

Example

The following expression is used in the code for the VDA Tool WzExport to determine if a given file can be written to.

... IF NOT WoCheckFile(file_name, /Write) THEN RETURN, '' ...
Keywords

*Layout_Name* — Specifies a string containing the name of the container form.

*Name* — Specifies a string containing the name of the button.

*Position* — A two-element array specifying the \( x, y \) coordinates of the color button in the bulletin board. (Default: \([0,0]\))

*Range* — A 2-element integer array specifying the range of color values in the color grid. The default range is \([0, (!D.Table_Size - 1)]\).

*Start_Value* — Specifies the color index to use as the original color for the color pixmap (integer).

*Title* — Specifies a string containing the title of the colortable dialog box.

Attachment Keywords

*Bottom* — If a widget ID is specified (for example, \( \text{Bottom}=\text{wid} \)), then the bottom of the button is attached to the top of the specified widget. If no widget ID is specified (for example, \( /\text{Bottom} \)), then the bottom of the button is attached to the bottom of the parent widget.

*Left* — If a widget ID is specified (for example, \( \text{Left}=\text{wid} \)), then the left side of the button is attached to the right side of the specified widget. If no widget ID is specified (for example, \( /\text{Left} \)), then the left side of the button is attached to the left side of the parent widget.

*Right* — If a widget ID is specified (for example, \( \text{Right}=\text{wid} \)), then the right side of the button is attached to the left side of the specified widget. If no widget ID is specified (for example, \( /\text{Right} \)), then the right side of the button is attached to the right side of the parent widget.

*Top* — If a widget ID is specified (for example, \( \text{Top}=\text{wid} \)), then the top of the button is attached to the bottom of the specified widget. If no widget ID is specified (for example, \( /\text{Top} \)), then the top of the button is attached to the top of the parent widget.

Discussion

This function is used in the Attributes dialog boxes of graphical VDA Tools.

---

**NOTE** Do not use WwSetValue with the *Userdata* keyword on the returned widget or on its parent widget after calling this routine. This use will cause unexpected side-effects, because WwSetValue changes the user data for a widget and for all of its children.
Contents of the Colortable Dialog Box

The Colortable dialog box has three main parts — the color palette area, the control area, and the action buttons.

Palette Area

An array of cells, one for each color index from the colormap. Numbers by the corners of the palette indicate the color index of the upper-left, upper-right, lower-left, and lower-right color square in the palette.

Controls Area

Selected Color — The index number of the currently selected color in the color table. Once a color is selected, the color pixmap next to the text field is updated. Select a color by clicking the left mouse button on a color cell in the palette of cells, or by entering the color’s index number in this text field and pressing <Return>.

Action Buttons

OK — Apply the selected color and exit the dialog box. Updates the color pixmap next to the color button.

Apply — Apply the selected color, but do not exit the dialog box. Updates the color pixmap next to the color button.

Cancel — Exit the dialog box; do not apply any color changes.

Help — Display online help on the dialog box.

Example

This example creates a color button inside a layout widget. The color button brings up a dialog box used to edit the current plot color. The name of the color button is specified in a resource file.

PRO OkApplyCB, wid, which
    ; ...
    ; Retrieve the currently selected color from the color button. The returned
    ; value will be an integer color index between 0 and 255.
    colorButtonWid = WwGetValue (wid, /Userdata)
    color = WoColorButtonGetValue (colorButtonWid)
    ; ...
END
PRO ChangeColorCB, wid, which
; ... 
; Set the color button color index. The new index should be an integer value in
; the range of the current plot colors.
  colorButtonWid = WwGetValue (wid, /Userdata)
  WoColorButtonSetValue, colorButtonWid, new_index
; ...
END

PRO CreateStuff
; ...
  top = WwInit (‘example’, ‘Example’, layout)
  ; Create a color button as a child of the layout widget. Color 5 will be
  ; displayed when the color button becomes visible. The title appears in the
  ; frame bar of the dialog that pops up when the user presses the color button.
  ; The button label is specified in a resource, as follows:
  *line_color_button.labelString: Line Color

  colorButtonWid = WoColorButton (layout, Title = ‘Line Color’, $ 
    Name = ‘line_color_button’, Start_value = 5)
  ; ...
  status = WwSetValue (top, /Display)
  WwLoop
END

See Also

WoColorButtonGetValue, WoColorButtonSetValue
WoColorButtonGetValue Function

Gets the currently selected color index from a color button created by WoColorButton.

**Usage**

color = WoColorButtonGetValue(wid)

**Input Parameters**

*wid* — Specifies the widget ID returned by WoColorButton (long).

**Returned Value**

*color* — The currently selected color index (integer).

**Keywords**

None.

**Example**

See the example given for WoColorButton.

**See Also**

WoColorButton, WoColorButtonSetValue
**WoColorButtonSetValue Function**

Sets the current color index for a color button created by WoColorButton, and updates the color button’s color pixmap.

**Usage**

\[
\text{color} = \text{WoColorButtonSetValue}(\text{wid}, \text{color})
\]

**Input Parameters**

- \textit{wid} — Specifies the widget ID returned by WoColorButton (long).
- \textit{color} — Specifies the new color index (integer).

**Keywords**

None.

**Example**

See the example given for WoColorButton.

**See Also**

WoColorButton, WoColorButtonGetValue

---

**WoColorConvert Function**

Convert from a long RGB value to an index into the current color table, or from an index in the current color table to an RGB value.

**Usage**

\[
\text{result} = \text{WoColorConvert}(\text{color})
\]

**Input Parameters**

- \textit{color} — The input RGB value or color index.
Returned Value

The returned value depends on the type of system and the keyword parameters that are used:

<table>
<thead>
<tr>
<th>System and Keyword</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit systems with either keyword</td>
<td>Return value is the same as the input color value.</td>
</tr>
<tr>
<td>24-bit system with /ColorToIndex keyword</td>
<td>Returns a color table index.</td>
</tr>
<tr>
<td>24-bit system with /IndexToColor keyword</td>
<td>Returns a long RGB value.</td>
</tr>
</tbody>
</table>

Keywords

*ColorToIndex* — If nonzero, converts an RGB value to a color table index.

*IndexToColor* — If nonzero, converts a color table index to an RGB value.

Discussion

WoColorConvert ensures that a selected color (e.g., in a VDA Tool application) looks the same on 8-bit and 24-bit displays.

The *ColorToIndex* and *IndexToColor* keywords are mutually exclusive.

The color conversion only occurs on 24-bit systems; the original color index is returned for other display types.

Example

The following PLOT command uses WoColorConvert to convert the plot color so that it works properly on a 24-bit display.

```pascal
color = 4
seed=2
PLOT, Randomn(seed,20), Color=WoColorConvert(color, /IndexToColor)
```
See Also

NOTE For detailed information on the following routine, please refer to the PV-WAVE Reference.

COLOR_CONVERT

WoColorGrid Function

Creates a grid of color squares from the current color table.

Usage

\[
\text{widget} = \text{WoColorGrid}(\text{parent})
\]

Input Parameters

\text{parent} — A string containing the widget ID of the parent widget.

Returned Value

\text{widget} — The widget ID of the color grid.

Keywords

\text{Title} — Specifies a string containing the title of the color grid.

\text{Model} — Specifies a string containing the color model used for the color grid (‘RGB’, ‘HLS’, or ‘HSV’).

\text{Range} — A 2-element integer array specifying the range of color values in the color grid. The default range is \([0, (!D.Table\_Size – 1)]\).

\text{Area} — If present and nonzero, the drawing area widget ID is returned.

\text{Num\_Columns} — An integer specifying the number of columns in the color grid. The default configuration is a square grid, where the number of rows and columns is equal.

\text{Cell\_Size} — A 2-element integer array specifying the size of each color cell, in pixels. The cells must be square. By default, the cells are 20-by-20 pixels.
Attachment Keywords

**Bottom** — If a widget ID is specified (for example, `Bottom=wid`), then the bottom of the color grid is attached to the top of the specified widget. If no widget ID is specified (for example, `/Bottom`), then the bottom of the color grid is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, `Left=wid`), then the left side of the color grid is attached to the right side of the specified widget. If no widget ID is specified (for example, `/Left`), then the left side of the color grid is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, `Right=wid`), then the right side of the color grid is attached to the left side of the specified widget. If no widget ID is specified (for example, `/Right`), then the right side of the color grid is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, `Top=wid`), then the top of the color grid is attached to the bottom of the specified widget. If no widget ID is specified (for example, `/Top`), then the top of the color grid is attached to the top of the parent widget.

Discussion

This function is used by the WzColorEdit VDA Tool and by the WoColorButton function.

Example

This example code displays a color grid with a button that allows you to cycle through different color hues.

```fortran
; This is the callback that is called when the “Cycle” or “Quit” button is pressed.
; If “Cycle” is pressed, the hues in the color grid are incremented so the
; colors will change. If the “Quit” button is pressed, the shell is destroyed.

PRO ButtonCB, wid, which
    DECLARE FUNC, WoColorGridGetValue
    ; Get the user data that was set earlier.
    cb_data = WwGetValue (wid, /Userdata)
    CASE which OF
        1: BEGIN                  ; Cycle
            ; Get the currently-displayed colors.
            colors = WoColorGridGetValue (cb_data.grid, 0, $
```
cb_data.num_colors)
; Add to the hues, so the colors will change. If the new values are greater
; than 360, set them back to a reasonable value by subtracting 360.
new_hues = colors(0, *) + cb_data.num_colors
    too_big_indices = WHERE(new_hues GE 360)
    ndim_too_big = SIZE(too_big_indices, /Ndim)
    IF ndim_too_big GT 0 THEN $
        new_hues(too_big_indices) = $
        new_hues(too_big_indices) - 360
    colors(0, *) = new_hues
; Change the colors displayed in the color grid.
WoColorGridSetValue, cb_data.grid, 0, colors
END
2: BEGIN                  ; Quit
    status = WwSetValue (cb_data.top, /Close)
END
ENDCASE
END
; This procedure demonstrates the use of the WoColorGrid,
; WoColorGridSetValue, and WoColorGridGetValue functions. It displays a
; shell window containing a color grid, and allows the user to cycle the color
; hues by pressing the "Cycle" button.
PRO WoColorGrid_test
    DECLARE FUNC, WoColorGrid
    ; Load the bottom of the color table with interesting colors.
    tek_color
    ; We’ll display 20 colors in the color grid.
    num_colors = 20
    ; Initialize Wave Widgets, and create the color grid and the push buttons.
    ; Note that we’re using HLS as the color model for the color grid.
    top = WwInit ('WoColorGrid_test', 'wocolorgrid_test', $
        layout, /Vertical, title = 'Color Grid Test')
    grid = WoColorGrid (layout, Title = 'Color Grid', $
        Model = 'HLS', $
        Range = [0, (num_colors - 1)])
    button_wid = WwButtonBox (layout, ['Cycle', 'Quit'], $
        'ButtonCB', buttons = buttons)
    ; Attach user data to the buttons, so we can access it from the button callback.
cb_data = {, top: top, grid: grid, num_colors: num_colors}
FOR i = 0, (N_ELEMENTS(buttons)-1) DO BEGIN
    status = WwSetValue (buttons(i), Userdata = cb_data)
ENDFOR
; Display the color grid.
status = WwSetValue (top, /Display)
WwLoop, /NoBlock
END

See Also
WoColorGridGetValue, WoColorGridSetValue

WoColorGridGetValue Function

Gets the color indices for a range of colors in a color grid.

Usage

colors = WoColorGridGetValue(wid, index, num_values)

Input Parameters

wid — Specifies the widget ID returned by WoColorGrid (long).
index — The first color index in the range of colors to be returned.
um_values — The number of color values to return.

Returned Value

colors — A 2D array of size 3-by-n, where n is either num_values or the number of color indices to the end of the color grid.

Keywords

None.
Discussion

The color values returned will be either RGB, HLS, or HSV values, depending on the currently active color model.

Example

See the example for WoColorGrid.

See Also

WoColorGrid, WoColorGridSetValue

WoColorGridSetValue Procedure

Sets the color indices for a range of colors in the color grid.

Usage

WoColorGridSetValue, wid, index, color

Input Parameters

wid — Specifies the widget ID returned by WoColorGrid (long).
index — The first color index in the range of colors to be set.
colors — An array of color table values from the currently active color model. The array structure must be one of the following:

A 3-element array if a single color is being set.

A 2D array of size 3-by-n, where n is the number of color values to set.

Keywords

None.

Example

See the example for WoColorGrid.
WoColorWheel Function

Creates a color wheel that can be used to modify a single color in the current color table.

Usage

\[ \text{wid} = \text{WoColorWheel}(\text{tool\_name}, \text{color\_index}, \text{value\_changed\_cb}) \]

Parameters

- **tool\_name** — A string containing the unique name of a VDA Tool.
- **color\_index** — The index of the color to display in the color wheel.
- **value\_changed\_cb** — A string containing the name of a callback procedure to call when the color value changes.

Keywords

None.

Returned Value

- **wid** — The widget ID of the color wheel.

Callback Parameters

The “value changed” callback must have the following two parameters:

- **tool\_name** — A string containing the unique name of a VDA Tool.
- **color\_index** — The index of the color displayed in the color wheel.

Discussion

The color wheel modifies the current color table directly. WoColorWheel is used by the WzColorEdit VDA Tool.
Example

This is the callback that is called when the color wheel value is changed.

PRO ValueChangedCB, tool_name, color_index
    print, ‘Value changed: ’, tool_name, color_index
END

This callback is called when the Color Wheel or Quit button is pressed. If Color Wheel is pressed, a color wheel appears for the selected color index. If the Quit button is pressed, the shell is destroyed.

PRO ButtonCB, wid, which
    DECLARE FUNC, TmGetAttribute
    DECLARE FUNC, WoColorWheel
    ; Get the tool name from user data.
    tool_name = WwGetValue (wid, /Userdata)
    CASE which OF
        1: BEGIN                  ; Color Wheel
            ; Get the selected color index.
            color_om = TmGetAttribute (tool_name, ‘TM’, ‘COLOR_OM’)
            index = WwGetValue (color_om) - 1
            ; Display the color wheel.
            wheel = WoColorWheel (tool_name, index, ‘ValueChangedCB’)
        END
        2: BEGIN                  ; Quit
            top = TmGetTop (tool_name)
            status = WwSetValue (top, /Close)
        END
    ENDCASE
END

This program demonstrates the WoColorWheel function. It creates a simple tool that allows the user to display the color wheel for any of the first 10 colors in the current color table.

PRO WoColorWheel_test
    DECLARE FUNC, TmGetUniqueToolName
    DECLARE FUNC, TmSetAttribute
; Initialize the tool manager.
TmInit
; Get a new unique tool name.

```
tool_name = TmGetUniqueToolName ('WoColorWheel_test')
; Initialize the top-level shell, and register the tool.
```

top = WwInit ('WoColorWheel_test', 'wocolorwheel_test', $
    layout, 'WoDestroyCB', Userdata = tool_name, $
    /Vertical, title = tool_name)

TmRegister, tool_name, top
; Create the color option menu and the button box.

```
color_menu = {, callback: 'TmNoOpCB', $
    button: '0', button: '1', button: '2', $
    button: '3', button: '4', button: '5', $
    button: '6', button: '7', button: '8', $
    button: '9', button: '10'}
```

color_om = WwOptionMenu (layout, 'Color Index', color_menu)

```
button_wid = WwButtonBox (layout, ['Color Wheel...', $
    'Quit'], 'ButtonCB', buttons = buttons)
```
; Save the attribute information needed for the callbacks.

```
status = TmSetAttribute (tool_name, 'TM', 'COLOR_OM', $
    color_om)
```

```
FOR i = 0, (N_ELEMENTS(buttons) - 1) DO BEGIN
    status = WwSetValue (buttons(i), Userdata = tool_name)
ENDFOR
; Display the tool.

```
status = WwSetValue (top, /Display)
```

WwLoop, /NoBlock

END
WoConfirmClose Procedure

Displays a dialog box requiring the user to confirm a window close action.

Usage

WoConfirmClose, wid, tool_name

Input Parameters

wid — (long) Specifies the widget ID of the parent of the dialog box.
tool_name — (string) Specifies the unique name of the VDA Tool to close.

Input Keywords

NoUnRegister — If nonzero, the VDA Tool is not automatically unregistered.

Output Keywords

Status — Returns the status of the user interaction: If 0, then the user cancelled the close operation; if 1, the user confirmed the close operation.

Discussion

If the user’s response is to close the VDA Tool, TmUnregister is called to close the specified VDA Tool.

This routine is registered with the ConfirmClose keyword of the WwInit and WwMainWindow functions. It is used when the Close command on the Window Manager menu is selected. This routine is also used by the File=>Exit command in VDA Tools.

WoConfirmClose checks the confirmClose resource to determine whether or not to display the confirmation dialog box. If this resource is set to FALSE, the TmUnregister procedure is called without asking the user to confirm the close.

Example 1

The following command is taken from the code for the WzColorEdit VDA Tool. Here, WoConfirmClose is called as a keyword to the WwMainWindow function.
... top = WwMainWindow (parent, layout, ‘WoDestroyCB’, $ Shell_name = ‘WzColorEdit’, Layout_name = ‘toolArea’, $ ConfirmClose = ‘WoConfirmClose’, Title = unique_name, $ /Form, Userdata = unique_name) ...

**Example 2**

The *NoUnRegister* and *Status* keywords are primarily used by the Navigator. These keywords allow the Navigator to systematically unregister VDA Tools when the Navigator is unregistered.

```
PRO NavCloseCB, wid, index
    ; Forward declaration of TM functions.
DECLARE FUNC, TmEnumerateToolNames
DECLARE FUNC, TmGetTop
    ; Make sure the user wants to quit.
top = TmGetTop(‘Navigator’)
WoConfirmClose, top, ‘Navigator’, Status=status, /NoUnRegister
    ; If answered in the affirmative, close all tools, ending with Nav.
IF status EQ 1 THEN BEGIN
    all_tools = TmEnumerateToolNames()
    FOR i=0, N_ELEMENTS(all_tools)-1 DO BEGIN
        IF all_tools(i) NE ‘Navigator’ $
            THEN TmUnregister, all_tools(i)
    ENDFOR
    ; Finish with the Navigator closed.
    TmUnregister, ‘Navigator’
ENDIF
END
```

**See Also**

WwInit, WwMainWindow, TmUnregister
WoDialogStatus Procedure

Saves or restores the status of a dialog box by saving or restoring the state of its widgets as stored in the Tools Manager.

Usage

WoDialogStatus, toolname, status

Input Parameters

toolname — (string) Specifies the unique name of the VDA Tool with which the dialog is associated.

status — Specifies a structure containing the status information. The fields are:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIDGET_ID</td>
<td>The widget ID of the widget to be saved or restored.</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>The default value to use upon restoring the dialog if nothing is found in the Tools Manager. In other words, this is the first time the dialog is invoked.</td>
</tr>
<tr>
<td>ITEM</td>
<td>The Tools Manager item under which to save or restore the setting.</td>
</tr>
<tr>
<td>ATTRIBUTE</td>
<td>The Tools Manager attribute under which to save or restore the setting.</td>
</tr>
</tbody>
</table>

Keywords

Restore — If nonzero, restores the status of the dialog box.

Save — If nonzero, saves the status of the dialog box.

Verbose — If nonzero, prints messages indicating the status of WoDialogStatus. This keyword is useful for debugging.

Discussion

The WIDGET_ID, DEFAULT, and ITEM fields must occur before the ATTRIBUTE field in the structure status.

WoDialogStatus can be used to save and restore the status of any WAVE Widget that allows WwSetValue and WwGetValue calls.
Example

The following code is taken from wzpreview.pro. It shows how WoDialogStatus is used to set and restore dialog box values.

FUNCTION WzPreviewDialogCB, wid, index
  DECLARE FUNC, TmGetAttribute
  DECLARE FUNC, TmSetAttribute
  ; Get the userdata and extract the tool name from the structure.
  setup_data = WwGetValue(wid, /Userdata)
  tool_name = setup_data.tool_name
  ; If OK or Apply was pressed, save the dialog. Let the WoGenericDialog
  ; routine handle Cancel and Help.
  CASE index OF
    1: BEGIN ; OK
      WoDialogStatus, tool_name, setup_data.status, /Save
      RETURN, 0
    END
    2: BEGIN ; Apply
      WoDialogStatus, tool_name, setup_data.status, /Save
      RETURN, 0
    END
    3: BEGIN ; Cancel
      RETURN, 0
    END
    4: BEGIN ; Help
      RETURN, 0
    END
  ENDCASE
END

PRO WzPreviewFreeDialog, tool_name
  DECLARE FUNC, TmGetTop
  DECLARE FUNC, TmGetAttribute
  parent = TmGetTop(tool_name)
  ; Create the elements of the dialog.
  helpfile= TmGetAttribute(tool_name, 'TM_HELP', 'HELP_FILE')
topic = TmGetMessage('wzpreview.ads', $ 'WzPreview_free_dialog_help')

title = TmGetMessage('wzpreview.ads', $ 'WzPreview_free_title') + tool_name

free_dialog = WoGenericDialog(parent, layout, $ 'WzPreviewDialogCB', $ Dialog_name = 'freeDialog', Title=title, $ Buttons=buttons, Help=[topic, helpfile], $ /Ok, /Apply, /Cancel)

; Create the elements of the dialog.

separator_label = WwText(layout, Text='', /Label,$ Name='separatorLabel', /Top, /Left)

separator_radio = WwRadioBox(layout, ['','','',''], $ /NOfMany, /Vertical, /Form, $ Layout_name='separatorForm', $ Name=['spaceToggle', 'commaToggle', $ 'tabToggle', 'otherToggle'], $ Toggles=separator_toggles, $ Top=separator_label, /Left)

other_text = WwText(separator_radio, Text='', $ Top=separator_toggles(2), $ Left=separator_toggles(3))

; Create data structure needed in the dialog callback.

status = {, $
    WIDGET_ID: separator_toggles(0), $
    DEFAULT: 1, $
    ITEM: 'TM', $
    ATTRIBUTE: 'SPACE_SEPARATOR', $
    WIDGET_ID: separator_toggles(1), $
    DEFAULT: 1, $
    ITEM: 'TM', $
    ATTRIBUTE: 'COMMA_SEPARATOR', $
    WIDGET_ID: separator_toggles(2), $
    DEFAULT: 1, $
    ITEM: 'TM', $
    ATTRIBUTE: 'TAB_SEPARATOR', $
    WIDGET_ID: separator_toggles(3), $
    DEFAULT: 0, $
    ITEM: 'TM', $}
WoFontOptionMenu Function

Creates an option menu with the standard list of software (vector-drawn) fonts found in PV-WAVE.

Usage

\[ \text{widget} = \text{WoFontOptionMenu}(\text{parent}, \text{toolname}) \]

Input Parameters

- \textit{parent} — Specifies the parent widget ID of the option menu (long).
**toolname** — (string) Specifies the unique name of the VDA Tool to which the option menu is to be attached.

**Returned Value**

**widget** — The widget ID of the container (long).

**Keywords**

**Name** — Specifies a string containing the name of the option menu.

**Start_Value** — Specifies a string containing the font command for the initially selected font.

**Attachment Keywords**

**Bottom** — If a widget ID is specified (for example, Bottom=wid), then the bottom of the option menu is attached to the top of the specified widget. If no widget ID is specified (for example, /Bottom), then the bottom of the option menu is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, Left=wid), then the left side of the option menu is attached to the right side of the specified widget. If no widget ID is specified (for example, /Left), then the left side of the option menu is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, Right=wid), then the right side of the option menu is attached to the left side of the specified widget. If no widget ID is specified (for example, /Right), then the right side of the option menu is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, Top=wid), then the top of the option menu is attached to the bottom of the specified widget. If no widget ID is specified (for example, /Top), then the top of the option menu is attached to the top of the parent widget.

**Discussion**

Software fonts, also called vector-drawn or Hershey fonts, are internal to PV-WAVE and are drawn with line vectors. Software fonts are specified by their font command. For example, the font command for the Simplex Greek font is !4
Example

This example creates a font option menu inside a layout widget. The default font on the menu is set to Duplex Roman.

```idl
PRO OkApplyCB, wid, which
    ...
    ; Retrieve the currently selected font from the font option menu. The returned
    ; value will be a Wave font string (i.e., '!3').
    fontWid = WwGetValue (wid, /Userdata)
    font = WoFontOptionmenuGetValue (fontWid)
    ...
END

PRO ChangeFontCB, wid, which
    ...
    ; Set the current value of the font option menu. The new font should be a
    ; PV-WAVE font string (i.e., '!3').
    fontWid = WwGetValue (wid, /Userdata)
    WoFontOptionmenuSetValue, fontWid, new_font
    ...
END

PRO CreateStuff
    ...
    top = WwInit ('example', 'Example', layout)
    ; Create a font option menu as a child of the layout widget. When the font is
    ; displayed, its value will be '!5' (Duplex Roman).
    fontWid = WoFontOptionmenu (layout, Start_value = '!5')
    ...
    status = WwSetValue (top, /Display)
    WwLoop
END
```

See Also

WoFontOptionMenuGetValue, WoFontOptionMenuSetValue
WoFontOptionMenuGetValue Function

Gets the software font command for the currently selected font.

**Usage**

\[ font = \text{WoFontOptionMenuGetValue}(wid) \]

**Input Parameters**

`wid` — Specifies the widget ID returned by WoFontOptionMenu (long).

**Returned Value**

`font` — A string containing the software font command for the currently selected font.

**Keywords**

None.

**Discussion**

Software fonts, also called vector-drawn or Hershey fonts, are internal to PV-WAVE and are drawn with line vectors. Software fonts are specified by their font command. For example, the font command for the Simplex Greek font is \[ \text{!4} \]

**Example**

See the example given for WoFontOptionMenu.

**See Also**

WoFontOptionMenu, WoFontOptionMenuSetValue
WoFontOptionMenuSetValue Procedure

Sets the current font and updates the font option menu.

Usage

WoFontOptionMenuSetValue, wid, font

Input Parameters

wid — Specifies the widget ID returned by WoFontOptionMenu (long).

font — Specifies a string containing the font command used to set the software font.

Keywords

None.

Discussion

This procedure also updates the font option menu (WoFontOptionMenu) by showing the font as selected.

Software fonts, also called vector-drawn or Hershey fonts, are internal to PV-WAVE and are drawn with line vectors. Software fonts are specified by their font command. For example, the font command for the Simplex Greek font is !4

Example

See the example given for WoFontOptionMenu.

See Also

WoFontOptionMenu, WoFontOptionMenuGetValue
WoGenericDialog Function

Creates a generic dialog box for use in VDA Tools.

Usage

container = WoGenericDialog(parent, topLayout [,callback])

Input Parameters

parent — Specifies the parent widget ID of the dialog (long).
callback — (optional) Specifies a string containing the name of the callback routine.

Output Parameters

topLayout — The widget ID of the top-level layout widget (long).

Returned Value

container — The widget ID of the dialog box (long).

Keywords

Apply — Inserts an action button with the label Apply in the dialog box.
Block — If nonzero, the dialog is blocking. (Default: nonblocking)
Board — If nonzero, a bulletin board layout is created for the topLayout. A form widget is the default.

Form widgets are “attached” to one another inside the layout. Bulletin board widgets are positioned in the layout with x, y coordinates.

Buttons — Returns the button widget IDs from WwButtonBox.
Cancel — Inserts an action button with the label Cancel in the dialog box. Unmanages the dialog when pressed.

Dialog_Name — Specifies a string containing a name for the generic dialog widget.

Dismiss — Inserts an action button with the label Dismiss in the dialog box.
**Help** — Specifies either a scalar string or a two-element string array, and inserts an action button with the label *Help* in the dialog box. This button is always inserted in the rightmost position in the action button area. If a scalar string, the button brings up help on the topic specified by the string. A two-element string array can be used to specify a topic name and the name of a help file.

**Next** — Inserts an action button with the label *Next* in the dialog box.

**No** — Inserts an action button with the label *No* in the dialog box. Unmanages the dialog when pressed.

**NoDestroy** — If nonzero, the dialog box widget is hidden instead of destroyed when any button that closes the dialog box (e.g., the *Cancel* button) is selected.

**Ok** — (The default) Inserts an action button with the label *OK* in the dialog box. Unmanages the dialog when pressed.

**Previous** — Inserts an action button with the label *Previous* in the dialog box.

**Reset** — Inserts an action button with the label *Reset* in the dialog box.

**Title** — Specifies a string containing the title of the dialog box.

**Yes** — (Overrides *Ok*) Inserts an action button with the label *Yes* in the dialog box. Unmanages the dialog when pressed.

**Callback Parameters**

- **wid** — The widget ID of the action area button.

- **index** — The index of the action area button that was pressed. The first button is 1; the second, 2; and so on.

**Discussion**

The purpose of this function is to provide an easy method for you to add a dialog box to a VDA Tool. This dialog is generic in that you can add widgets of your choice to it.

The dialog is returned unmanaged; it is assumed that the calling routine will fill the upper layout and then manage the dialog.

Buttons that will appear at the bottom of the dialog are controlled with keyword parameters. The buttons will be positioned properly for compliance with the Motif style guide.
This function assumes that most dialog attributes will be controlled via resources rather than a parameter list. Use the Resource keyword on WwInit, or the WtResource function, to set the resource values.

The behavior of the dialog box is as follows:

- The second parameter returns the widget ID of the layout widget, which can be filled in by the caller. This layout widget will always be a form widget.
- The optional “callback” (third) parameter supplies a callback that will be called when a button is selected. This callback behaves the same as a callback from the WwGenericDialog function.

The callback should be a function that returns 0 (accept default dismiss behavior) or 1 (veto dismiss).

- If the Help keyword is used, the rightmost (last) button is assumed to be the Help button (no matter what its name or label), and it will bring up the online Help system. You can specify a help file other than the standard one for PV-WAVE by passing a 2-element string array ['help_topic_name', 'filename'] with the Help keyword.

Example

This example creates an entry point from which you can bring up a dialog box created with WogenericDialog. The dialog box contains three action buttons and a text field.

FUNCTION Wogenericdialog_btn_cb, wid, data
    ; This function is called when any button in the generic dialog is pressed. If the ; function returns 0, the default button action (close for OK and Cancel, don't ; close for Apply) happens. If a 1 is returned for the OK or Cancel button ; presses, the dialog will not be closed. This allows the programmer to keep ; the dialog up if an error condition needs to be corrected.
    PRINT, 'wogenericdialog_btn_cb', wid, data
    RETURN, 0
END

; This procedure prints what the user typed into the text field.

PRO Text_cb, wid, data
    value = WwGetValue (wid)
    print, 'Value: ', value
END

PRO wogenericdialog_button_cb, wid, data
    ; This procedure creates a generic dialog containing three buttons: "OK", "Cancel",
WoGenericDialog Function

"Apply", and "Cancel". The dialog contains a text widget to type into.

```wotext
; COMMON wogenericdialog_common, topshell, $ wogenericdialog_shell
CASE data OF
  1: BEGIN ; Simple
dialog_wid = wogenericdialog (topshell, layout, $ /OK, /Apply, /Cancel)
text_wid = WwText (layout, 'Text_CB', Label = $ 'Type something here: ', /Top, /Left)
status = WwSetValue (dialog_wid, /Show)
END
  2: BEGIN ; Quit
status = WwSetValue (topshell, /Close)
END
ENDCASE
END
```

; This is the main entry point. It creates a window with two buttons; one
; button is used to create a dialog with WoGenericDialog(), and the other is
; used to quit the sample application.

```wotext
PRO wogenericdialog_test
COMMON wogenericdialog_common, topshell, $ wogenericdialog_shell
topshell = wwinit ('wogenericdialog_test', 'Appl', $ workarea)
buttonbox = WwButtonBox (workarea, ['Dialog...','Quit'], $ 'wogenericdialog_button_cb', /horizontal)
status = wwssetvalue(topshell, /display)
WwLoop
END
```

See Also

WwGenericDialog
WoGetToolNameFromTitle Function

Gets the unique name of a VDA tool given the unique window title of the VDA Tool.

Usage

\[ tool\_name = \text{WoGetToolNameFromTitle}(\text{window}\_title) \]

Input Parameters

\text{window}\_title — A string specifying a unique window title of a VDA Tool.

Keywords

None.

Example

The following code allows selection of tools from a list of tool titles.

```
; Main procedure:
; Get the names of all tools and their titles.
tools = TmEnumerateToolNames(Titles=titles)

; Sort the titles alphabetically.
titles = titles(SORT(titles))

; Create a list containing the titles.
title_list = WwList(layout, titles, /Multi, $ /Top, /Bottom, /Right, /Left, Name='titleList')

; Callback procedure:
; Get the titles of selected tools.
titles = WwGetValue(title_list)

; Get the tool names from the titles.
tool_names = WoGetToolNameFromTitle(titles)
```
WoGetUniqueWindowTitle Function

Given two descriptive strings, generates a unique window title for a VDA Tool. Given a window title, adds a numeric suffix to make the title unique.

Usage

\[ name = \text{WoGetUniqueWindowTitle}(primary, secondary) \]

Input Parameters

- **primary** — A string specifying the primary component of the window title, typically a PV=WAVE variable name.
- **secondary** — A string specifying the secondary component. Typically the VDA Tool name, appears after a hyphen (–).

Returned Value

- **name** — A string containing a unique name for the window.

Keywords

None.

Discussion

If the window “MyImage – Image” already exists, this routine returns “MyImage – Image – (1)”.

Example

The following procedure creates and sets the title of a VDA Tool.

```pascal
PRO WzToolNameSetTitle, tool_name
  DECLARE FUNC, TmEnumerateVars
```
; Get the description of this tool, i.e. 'Surface Tool'.
tool_title = TmGetMessage('WzToolName_title')

; Get the variable name(s).
var_names = TmEnumerateVars(tool_name)
IF N_ELEMENTS(var_names) EQ 1 THEN vars = var_names(0) $
ELSE vars = STRJOIN(var_names, ',')

; Create the window title and set it.
window_title = WoGetUniqueWindowTitle(vars, tool_title)
WoSetWindowTitle, tool_name, window_title
END

See Also
WoSetWindowTitle

WoLabeledText Function

Creates a group of aligned text widgets (widgets with a label and a text field).

Usage

\[ \text{widget} = \text{WoLabeledText}(\text{parent}, \text{label\_names}, \text{verify\_callback}) \]

Input Parameters

- **parent** — Specifies the parent widget ID of the text widget (long).
- **label\_names** — Specifies an array of strings containing the names of the label widgets.
- **verify\_callback** — Specifies a string containing the name of the text verification callback.

Returned Value

- **widget** — The widget ID of the container widget (long).
**Input Keywords**

*Cols* — Specifies the number of columns in the text widgets. (Default: 10)

*Horizontal* — If nonzero, aligns the text widgets horizontally. (Default: vertical alignment)

*Layout_Name* — Specifies a string containing the name of the container widget.

*Position* — A two-element array specifying the x, y coordinates of the text widgets in the bulletin board. (Default: [0,0])

*Text_Names* — Specifies a string array containing the names of the text widgets. (Default: ['text_0', 'text_1', ...])

*Text_Strings* — Specifies a string array containing the initial contents of the text widgets.

**Output Keywords**

*Text_Widgets* — An array of widget IDs for the text widgets.

**Attachment Keywords**

*Bottom* — If a widget ID is specified (for example, Bottom=wid), then the bottom of the text widget is attached to the top of the specified widget. If no widget ID is specified (for example, /Bottom), then the bottom of the text widget is attached to the bottom of the parent widget.

*Left* — If a widget ID is specified (for example, Left=wid), then the left side of the text widget is attached to the right side of the specified widget. If no widget ID is specified (for example, /Left), then the left side of the text widget is attached to the left side of the parent widget.

*Right* — If a widget ID is specified (for example, Right=wid), then the right side of the text widget is attached to the left side of the specified widget. If no widget ID is specified (for example, /Right), then the right side of the text widget is attached to the right side of the parent widget.

*Top* — If a widget ID is specified (for example, Top=wid), then the top of the text widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, /Top), then the top of the text widget is attached to the top of the parent widget.
**Discussion**

The number of text fields is determined by the number of elements in the `label_names` parameter.

The text fields are vertically-aligned unless the *Horizontal* keyword is specified.

**Example**

This example creates a group of vertically aligned text widgets. The text field labels can be obtained from a resource file.

```pro
PRO ValueChangedCB, wid, which
...
; This callback is called if the user types <Return> within one of the text wi
; gets. The value will be a PV-WAVE string, identical to the value returned from
; WwText.
value = WwGetValue (wid)
   print, value
...
END
PRO CreateStuff
...
top = WwInit ('example', 'Example', layout)
; Create a group of vertically aligned text widgets. The label strings will be
; blank unless they are loaded from a resource file or the X resource database.
; The label strings should be specified in resources, such as:
*distance.labelString: Distance
*velocity.labelString: Velocity
*acceleration.labelString: Acceleration
textWids = WoLabeledText (layout, ['distance', 'velocity', $
    'acceleration'], 'ValueChangedCB')
...
status = WwSetValue (top, /Display)
WwLoop
END
```

**See Also**

WwInit
WoLinestyleOptionMenu Function

Creates an option menu for selecting linestyles.

Usage

\[
\text{widget} = \text{WoLinestyleOptionMenu}(\text{parent}, \text{toolname})
\]

Input Parameters

\text{parent} — Specifies the parent widget ID of the option menu (long).

\text{toolname} — (string) Specifies the unique name of the VDA Tool to which the option menu is to be attached.

Returned Value

\text{widget} — The widget ID of the option menu (long).

Keywords

\text{Nolinestyle} — Adds a \text{None} option to the linestyles option menu.

\text{Start_Value} — Specifies a string containing the index of the initially selected linestyle.

Attachment Keywords

\text{Bottom} — If a widget ID is specified (for example, \text{Bottom}=\text{wid}), then the bottom of the option menu is attached to the top of the specified widget. If no widget ID is specified (for example, \text{/Bottom}), then the bottom of the option menu is attached to the bottom of the parent widget.

\text{Left} — If a widget ID is specified (for example, \text{Left}=\text{wid}), then the left side of the option menu is attached to the right side of the specified widget. If no widget ID is specified (for example, \text{/Left}), then the left side of the option menu is attached to the left side of the parent widget.

\text{Right} — If a widget ID is specified (for example, \text{Right}=\text{wid}), then the right side of the option menu is attached to the left side of the specified widget. If no widget ID is specified (for example, \text{/Right}), then the right side of the option menu is attached to the right side of the parent widget.
**Top** — If a widget ID is specified (for example, Top=wid), then the top of the option menu is attached to the bottom of the specified widget. If no widget ID is specified (for example, /Top), then the top of the option menu is attached to the top of the parent widget.

**Discussion**

The available linestyles and their index numbers are listed in the following table:

<table>
<thead>
<tr>
<th>Index</th>
<th>X Windows Style</th>
<th>Microsoft Windows Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Solid</td>
<td>Solid</td>
</tr>
<tr>
<td>1</td>
<td>Dotted</td>
<td>Short dashes</td>
</tr>
<tr>
<td>2</td>
<td>Dashed</td>
<td>Long dashes</td>
</tr>
<tr>
<td>3</td>
<td>Dash dot</td>
<td>Long-short dashes</td>
</tr>
<tr>
<td>4</td>
<td>Dash-dot-dot-dot</td>
<td>Long-short-short dashes</td>
</tr>
<tr>
<td>5</td>
<td>Long dashes</td>
<td>Long dashes</td>
</tr>
</tbody>
</table>

**Example**

This example creates a linestyle option menu inside a layout widget. The default linestyle on the menu is set to dash-dot.

```plaintext
PRO OkApplyCB, wid, which

; ...;
; Retrieve the currently selected linestyle from the option menu. The returned;
; value will be an integer between 0 and 5 that corresponds to the setting of;
; the !P.Linestyle system variable.
linestyleWid = WwGetValue (wid, /Userdata)
linestyle = WoLinestyleOptionMenuGetValue (linestyleWid)
; ...;
END

PRO ChangeLinestyleCB, wid, which

; ...;
; Set the linestyle in the option menu. The new value should be an integer;
; between 0 and 5; see !P.Linestyle for more details.
linestyleWid = WwGetValue (wid, /Userdata)
```
WoLineStyleOptionMenuGetValue Function

Gets the currently selected linestyle.

Usage

\[
\text{linestyle} = \text{WoLineStyleOptionMenuGetValue}(\text{wid})
\]

Input Parameters

\(\text{wid}\) — Specifies the widget ID returned by WoLineStyleOptionMenu (long).

Returned Value

\(\text{linestyle}\) — The linestyle index (integer).

Keywords

None.
Discussion

The available linestyles and their index numbers are listed in the following table:

<table>
<thead>
<tr>
<th>Index</th>
<th>X Windows Style</th>
<th>Microsoft Windows Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Solid</td>
<td>Solid</td>
</tr>
<tr>
<td>1</td>
<td>Dotted</td>
<td>Short dashes</td>
</tr>
<tr>
<td>2</td>
<td>Dashed</td>
<td>Long dashes</td>
</tr>
<tr>
<td>3</td>
<td>Dash dot</td>
<td>Long-short dashes</td>
</tr>
<tr>
<td>4</td>
<td>Dash-dot-dot-dot</td>
<td>Long-short-short dashes</td>
</tr>
<tr>
<td>5</td>
<td>Long dashes</td>
<td>Long dashes</td>
</tr>
</tbody>
</table>

Example

See the example given for WoLinestyleOptionMenu.

See Also

WoLinestyleOptionMenu, WoLinestyleOptionMenuSetValue

WoLinestyleOptionMenuSetValue Procedure

Sets the option menu to a specified linestyle.

Usage

WoLineStyleOptionMenuSetValue, wid, linestyle

Input Parameters

- **wid** — (long) Specifies the widget ID returned by WoLinestyleOptionMenu.
- **linestyle** — (integer) Specifies the index value of the linestyle to set.

Keywords

None.
**Discussion**

The available linestyles and their index numbers are listed in the following table:

<table>
<thead>
<tr>
<th>Index</th>
<th>X Windows Style</th>
<th>Microsoft Windows Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Solid</td>
<td>Solid</td>
</tr>
<tr>
<td>1</td>
<td>Dotted</td>
<td>Short dashes</td>
</tr>
<tr>
<td>2</td>
<td>Dashed</td>
<td>Long dashes</td>
</tr>
<tr>
<td>3</td>
<td>Dash dot</td>
<td>Long-short dashes</td>
</tr>
<tr>
<td>4</td>
<td>Dash-dot-dot-dot</td>
<td>Long-short-short dashes</td>
</tr>
<tr>
<td>5</td>
<td>Long dashes</td>
<td>Long dashes</td>
</tr>
</tbody>
</table>

**Example**

See the example given for WoLinestyleOptionMenu.

**See Also**

WoLinestyleOptionMenu, WoLinestyleOptionMenuGetValue

---

**WoLoadResources Procedure**

Loads resources and strings from a file for VDA tools.

**Usage**

WoLoadResources, *file*

**Input Parameters**

*file* — The name of the resource file to be loaded.

**Returned Value**

None.
Keywords

**Appdir** — A string that specifies the application directory name. This is the directory in which the application searches for resource files, string resource files, and icon files. See the Discussion. (Default: vdatools)

**Subdir** — A string specifying a resource file subdirectory. See the Discussion.

Discussion

**UNIX USERS** By default, the function looks for `file` first in directories specified by the environment variable `WAVE_RESPATH`. This environment variable is a colon separated list of directories, similar to the PV-WAVE `WAVE_PATH` environment variable. If not found in a `WAVE_RESPATH` directory, the directory `$WAVE_DIR/xres/!Lang/vdatools` is searched, where `!Lang` represents the value of the `!Lang` system variable in PV-WAVE.

**OpenVMS USERS** By default, the function looks for `file` first in directories specified by the logical `WAVE_RESPATH`. This logical is a comma separated list of directories and text libraries, similar to the OpenVMS `WAVE_PATH` logical. If not found in a `WAVE_RESPATH` directory, the directory `WAVE_DIR:[XRES.!Lang.VDATOOLS]` is searched, where `!Lang` represents the value of the `!Lang` system variable in PV-WAVE.

**Windows USERS** By default, the function looks for `file` first in directories specified by the environment variable `WAVE_RESPATH`. This environment variable is a semicolon separated list of directories, similar to the PV-WAVE `WAVE_PATH` environment variable. If not found in a `WAVE_RESPATH` directory, the directory `%WAVE_DIR%\xres\!Lang\vdatools` is searched, where `!Lang` represents the value of the `!Lang` system variable in PV-WAVE.

If `Subdir` is specified, the file is searched for in:

(UNIX)  `<wavedir>/xres/subdir/vdatools`

(OpenVMS)  `<wavedir>:[XRES.SUBDIR.VDATOOLS]`

(Windows)  `<wavedir>\xres\subdir\vdatools`

Where `<wavedir>` is the main PV-WAVE directory.

If `Appdir` is specified, the application searches for resources in the following directory:
WoLoadResources Procedure

(UNIX)  
< wavedir>/xres/!Lang/appdir  

(OpenVMS)  
< wavedir>: [XRES.!Lang.APPDIR]  

(Windows)  
< wavedir>/xres/!Lang/appdir  

Where < wavedir > is the main PV-WAVE directory.

If Subdir and Appdir are specified, the application searches for resources in the following directory:

(UNIX)  
< wavedir>/xres/subdir/appdir  

(OpenVMS)  
< wavedir>: [XRES.SUBLDIR.APPDIR]  

(Windows)  
< wavedir>/xres/subdir/appdir  

Where < wavedir > is the main PV-WAVE directory.

If the file to be loaded is not already in the resource database, it is loaded and added to the resource database list of files.

**NOTE**  
WoLoadResources keeps a list of the loaded files so that files aren’t redundantly loaded.

The first time WoLoadResources is invoked, it loads the file wzglobal.ad.

**Example**

These calls load resources and strings for the Printer Setup dialog box used in VDA Tools. This code was taken from the program woprintsetup.pro. The resource and string files are located in:

(UNIX)  
< wavedir>/xres/american/vdatools  

(OpenVMS)  
< wavedir>: [XRES.AMERICAN.VDATOOLS]  

(Windows)  
< wavedir>/xres/american/vdatools  

Where < wavedir > is the main PV-WAVE directory.

WoLoadResources, ‘woprintsetup.ad’  
; Load resources.

WoLoadStrings, ‘woprintsetup.ads’  
; Load the strings.

**See Also**

WoLoadStrings,  WoBuildResourceFilename
WoLoadStrings Procedure

Loads strings from a resource file for use by the VDA tools.

Usage

WoLoadStrings, file

Input Parameters

file — The name of the file to be loaded.

Returned Value

None.

Keywords

Appdir — A string that specifies the application directory name. This is the directory in which the application searches for resource files, string resource files, and icon files. See the Discussion. (Default: vdatools)

Subdir — Specifies a subdirectory in which to look for the resource file.

Discussion

UNIX USERS  By default, the function looks for file first in directories specified by the environment variable WAVE_RESPATH. This environment variable is a colon separated list of directories, similar to the PV-WAVE WAVE_PATH environment variable. If not found in a WAVE_RESPATH directory, the directory $WAVE_DIR/xres/!Lang/vdatools is searched, where !Lang represents the value of the !Lang system variable in PV-WAVE.

OpenVMS USERS  By default, the function looks for file first in directories specified by the logical WAVE_RESPATH. This logical is a comma separated list of directories and text libraries, similar to the OpenVMS WAVE_PATH logical. If not found in a WAVE_RESPATH directory, the directory WAVE_DIR:[XRES.!Lang.VKTOOLS] is searched, where !Lang represents the value of the !Lang system variable in PV-WAVE.
**Windows USERS**  By default, the function looks for *file* first in directories specified by the environment variable *WAVE_RESPATH*. This environment variable is a semicolon separated list of directories, similar to the PV-WAVE *WAVE_PATH* environment variable. If not found in a *WAVE_RESPATH* directory, the directory %WAVE_DIR%\xres\!Lang\vdatools is searched, where !Lang represents the value of the !Lang system variable in PV-WAVE.

If *Subdir* is specified, the file is searched for in:

(UNIX)  <wavedir>/xres/subdir/vdatools

(OpenVMS)  <wavedir>:[XRES_SUBDIR.VDATOOLS]

(Windows)  <wavedir>\xres\subdir\vdatools

Where <wavedir> is the main PV-WAVE directory.

If *Appdir* is specified, the application searches for resources in the following directory:

(UNIX)  <wavedir>/xres/!Lang/appdir

(OpenVMS)  <wavedir>:[XRES.!Lang.APPDIR]

(Windows)  <wavedir>\xres\!Lang\appdir

Where <wavedir> is the main PV-WAVE directory.

If *Subdir* and *Appdir* are specified, the application searches for resources in the following directory:

(UNIX)  <wavedir>/xres/subdir/appdir

(OpenVMS)  <wavedir>:[XRES_SUBDIR.APPDIR]

(Windows)  <wavedir>\xres\subdir\appdir

Where <wavedir> is the main PV-WAVE directory.

If the file to be loaded is not already in the resource database, it is loaded and added to the resource database list of files.

This procedure functions as a wrapper to the WoLoadResources procedure with the *Strings* keyword set.

**Example**

These calls load resources and strings for the Printer Setup dialog box used in VDA Tools. This code was taken from the program woprintsetup.pro. The resource and string files are located in:
WoLoadResources, 'woprintsetup.ad'
   ; Load resources.
WoLoadStrings, 'woprintsetup.ads'
   ; Load the strings.

See Also
WoLoadResources

**WoMenuBar Function**

Create a menu bar for a VDA Tool.

**Usage**

```
bar = WoMenuBar(parent, toolname [,menus])
```

**Input Parameters**

- **parent** — (long) Specifies the parent widget ID of the menu bar.
- **toolname** — (string) Specifies the unique name of the VDA Tool to which the menu bar is to be attached.
- **menus** — (optional) Specifies an unnamed structure containing menu definitions. This parameter is only needed if you do not wish to use the standard graphics menu bar (i.e., when the **Graphics** keyword is not specified).

**Returned Value**

- **bar** — The widget ID of the menu bar.

**Keywords**

- **Graphics** — If nonzero, specifies that the standard menu for graphics VDA Tools should be used.
Attachment Keywords

**Bottom** — If a widget ID is specified (for example, `Bottom=wid`), then the bottom of the menu bar is attached to the top of the specified widget. If no widget ID is specified (for example, `/Bottom`), then the bottom of the menu bar is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, `Left=wid`), then the left side of the menu bar is attached to the right side of the specified widget. If no widget ID is specified (for example, `/Left`), then the left side of the menu bar is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, `Right=wid`), then the right side of the menu bar is attached to the left side of the specified widget. If no widget ID is specified (for example, `/Right`), then the right side of the menu bar is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, `Top=wid`), then the top of the menu bar is attached to the bottom of the specified widget. If no widget ID is specified (for example, `/Top`), then the top of the menu bar is attached to the top of the parent widget.

Discussion

If the **Graphics** keyword is specified, a predefined set of standard menus is provided automatically. These predefined menus are equipped with functional callbacks. If you choose to use the predefined set of menus in your application, you do not need to modify the underlying structure of the menus or the callbacks. Both the predefined callbacks and the underlying menu structures are defined in the following files in the `vdatools` subdirectory of the Standard Library:

- `wographicsmenus.pro` — Uses unnamed structures to define the menus and callbacks for the menus.
- `wographicsmenuscb.pro` — Contains the callback routines for each menu.

If you do not choose to use the standard menu bar, then you can use these files as templates for creating a customized menu bar. Use the optional `menu` parameter to specify the name of the unnamed structure containing the new menu definitions. The structure definition can be added to the VDA Tool program with an `@include` statement or included in the VDA Tool program itself.

For a customized menu bar, you will also have to write and include appropriate callback procedures in your application.
NOTE For information on how to use the functions on the standard menu bar, run one of the graphical VDA Tools, such as WzPlot, and read about the menu bar in online help.

Example 1
In this example, the *Graphics* keyword is used to place a standard menu bar in a VDA Tool.

```
bar = WoMenuBar(layout, tool_name, /Graphics, /Top, /Left, /Right)
```

Example 2
This example shows a portion of the resource file for the WzPreview VDA Tool. The resources shown are used by the menus of this VDA Tool. These menus differ from the standard menu bar. The menu bar structure is then defined.

```
wzpreview.ad - resources for the WzPreview menus

! Menus and menu items
!
! Edit items
!
*WzPreview*EditDefineHeader.labelString: Define Header...
*WzPreview*EditDefineHeader.mnemonic: H
*WzPreview*EditDefineHeader.acceleratorText: Ctrl+H
*WzPreview*EditDefineHeader.accelerator: Ctrl<Key>H

*WzPreview*EditDefineRecord.labelString: Define Record...
*WzPreview*EditDefineRecord.mnemonic: R
*WzPreview*EditDefineRecord.acceleratorText: Ctrl+R
*WzPreview*EditDefineRecord.accelerator: Ctrl<Key>R

*WzPreview*EditDefineField.labelString: Define Field...
*WzPreview*EditDefineField.mnemonic: F
*WzPreview*EditDefineField.acceleratorText: Ctrl+F
*WzPreview*EditDefineField.accelerator: Ctrl<Key>F
```
WoMenuBar Function

*WzPreview*EditSelectArea.labelString: Select Area
*WzPreview*EditSelectArea.mnemonic: S
*WzPreview*EditClear.labelString: Clear
*WzPreview*EditClear.mnemonic: C
*WzPreview*EditClear.acceleratorText: Ctrl+X
*WzPreview*EditClear.accelerator: Ctrl<Key>X

*WzPreview*EditClearAll.labelString: Clear All
*WzPreview*EditClearAll.mnemonic: A
*WzPreview*EditClearAll.acceleratorText: Ctrl+A
*WzPreview*EditClearAll.accelerator: Ctrl<Key>A

Attributes items

*WzPreview*AttributesSetup.labelString: Setup...
*WzPreview*AttributesSetup.mnemonic: S
*WzPreview*AttributesVariable.labelString: Variable
*WzPreview*AttributesVariable.mnemonic: r

*WzPreview*AttributesVariable*NoVariable.labelString: No variables

*WzPreview*AttributesFixedFormat.labelString: Fixed Format...
*WzPreview*AttributesFixedFormat.mnemonic: x

*WzPreview*AttributesFreeFormat.labelString: Free Format...
*WzPreview*AttributesFreeFormat.mnemonic: r

; Define the WzPreview menu bar structure (in the wzpreview.pro procedure).

wzpreview_menus = {
  NAME: ['FileMenu', 'FileMenu'],
  MENUBUTTON: '',
  MENU: {
    CALLBACK: 'WoGMBFileOpenCB',
    NAME: 'FileOpen',
    BUTTON: '',
  },}
CALLBACK: 'WzPreviewFileSaveCB', $ 
NAME: 'FileSave', $ 
BUTTON: '', $ 
CALLBACK: 'WzPreviewFileSaveAsCB', $ 
NAME: 'FileSaveAs', $ 
BUTTON: '', $ 
CALLBACK: 'WoGMBFileSaveTemplateAsCB', $ 
NAME: 'FileSaveTemplateAs', $ 
BUTTON: '', $ 
SEPARATOR: 1, $ 
CALLBACK: 'WoGMBFileExportVariableCB', $ 
NAME: 'FileExportVariable', $ 
BUTTON: '', $ 
SEPARATOR: 1, $ 
NAME: 'FileGenerateCode', $ 
CALLBACK: 'WoGMBFileGenerateCodeCB', $ 
BUTTON: '', $ 
SEPARATOR: 1, $ 
CALLBACK: 'WoGMBFileCloseCB', $ 
NAME: 'FileClose', $ 
BUTTON: '' $ 
}, $ 
NAME: ['EditMenu', 'EditMenu'],$ 
MENUBUTTON: '',$
MENU:{,}$ 
CALLBACK: 'WzPreviewEditCB', $ 
NAME: 'EditDefineHeader',$
BUTTON: '',$
NAME: 'EditDefineRecord',$
BUTTON: '',$
NAME: 'EditDefineField',$
BUTTON: '',$
SEPARATOR: 1, $
NAME: 'EditSelectArea',$

BUTTON: '', $  
SEPARATOR: 1, $  
NAME: 'EditClear', $  
BUTTON: '', $  
NAME: 'EditClearAll', $  
BUTTON: '' $  
}, $  
NAME: ['AttributesMenu', 'AttributesMenu'], $  
MENUBUTTON: '', $  
MENU:{, $  
  CALLBACK: 'WzPreviewSetupCB', $  
  NAME: 'AttributesSetup', $  
  BUTTON: '', $  
  NAME: ['AttributesVariable', $  
  'AttributesVariable'], $  
  MENUBUTTON: '', $  
  MENU:{, $  
    CALLBACK: 'WzPreviewAttributesVarCB', $  
    NAME: 'NoVariables', $  
    BUTTON: '' $  
  }, $  
}, $  
SEPARATOR: 1, $  
CALLBACK:'WzPreviewFormatCB',$  
NAME: 'AttributesFixedFormat',$  
TOGGLE:'',$  
NAME: 'AttributesFreeFormat',$  
TOGGLE:''$  
}, $  
NAME: ['WindowMenu', 'WindowMenu'],$  
MENUBUTTON: '', $  
MENU:{,CALLBACK:'WoGMBWindowCB',$  
  NAME: 'Window_1',$  
  BUTTON: '' $  
}, $
CALLBACK: ‘WoGMBHelpCB’, $
NAME: ['HelpMenu', 'HelpMenu'],$
MENUBUTTON: ' ', $
MENU: {, CALLBACK: 'WoGMBHelpOnWindowCB', $
   NAME: 'HelpOnWindow', $
   BUTTON: ' ', $
   CALLBACK: 'WoGMBHelpIndexCB', $
   NAME: 'HelpIndex', $
   BUTTON: ' ', $
   CALLBACK: 'WoGMBHelpOnHelpCB', $
   NAME: 'HelpOnHelp', $
   BUTTON: ' ', $
   SEPARATOR: 1, $
   CALLBACK: 'WoGMBHelpOnVersionCB', $
   NAME: 'HelpOnVersion', $
   BUTTON: ' ' $
}

; Use the structure with customized menus, but include the Graphics keyword
; to include callbacks for the graphics menu items.
bar = WoMenuBar(layout, tool_name, wzpreview_menus, $
   /Graphics, /Top, /Left, /Right)

; If no filename or a bad file was specified, make all the
; filename-dependent menu items insensitive.
IF STRLEN(file) EQ 0 THEN BEGIN
   panes = [1, $  ; File menu
   1, $
   1, $
   1, $
   1, $
   2, $  ; Edit menu
   2, $
   2, $
   2, $
   2, $
WoMenuBarSetSensitivity Procedure

Sets the sensitivity of one or more items in a menu.

Usage

WoMenuBarSetSensitivity, toolname, pane_index, item_index, sensitivity

Input Parameters

- **toolname** — (string) Specifies the unique name of the VDA Tool to which the menu bar is attached.
- **pane_index** — Specifies the index for a specific menu pane in the array of menu panes. The first menu is 1; the second, 2; and so on.

See Also

WoMenuBarSetSensitivity

WoMenuBarSetSensitivity Procedure

Sets the sensitivity of one or more items in a menu.

Usage

WoMenuBarSetSensitivity, toolname, pane_index, item_index, sensitivity

Input Parameters

- **toolname** — (string) Specifies the unique name of the VDA Tool to which the menu bar is attached.
- **pane_index** — Specifies the index for a specific menu pane in the array of menu panes. The first menu is 1; the second, 2; and so on.
NOTE A submenu’s index number is the number following its parent menu index.

item_index — Specifies the index of the menu item to change in the specified menu pane. The first item is 1; the second, 2; and so on.

NOTE Separators are counted as items in the menu, and must be included in the item_index specification.

sensitivity — Specifies a scalar or array with the sensitivity of the menu item or items. A value of 0 makes the item insensitive; 1 makes the item sensitive.

Keywords
None.

Discussion
If the sensitivity parameter is a scalar, then all menu items are set to the value of sensitivity. If the sensitivity parameter is an array, each menu item is set to its corresponding element in sensitivity. In other words, the sensitivity of the first item is set to the first value in the array, the second item to the second value, and so on. If sensitivity is an array with fewer elements than menu items, the extra items are set to 1 (sensitive).

Example
For an example using WoMenuBarSetSensitivity, see WoMenuBar.

See Also
WoMenuBar
WoMenuBarSetToggle Procedure

Sets the status of a menu toggle button.

Usage

WoMenuBarSetToggle, tool_name, pane_index, item_index, value

Input Parameters

*tool_name* — A string specifying the unique name of the VDA Tool to which the menu bar is attached.

*pane_index* — An integer or array of integers specifying the index for a specific menu pane in the array of menu panes. The first menu is 1; the second, 2; and so on.

*item_index* — An integer or array of integers specifying the index of the menu item to change in the specified menu pane. The first item is 1; the second, 2; and so on. This parameter must have the same number of elements as *pane_index*.

*value* — An integer or array of integers specifying the status of the toggle. Selected = 1; deselected = 0.

Keywords

*Item* — A string specifying a menubar item as specified in WoMenuBar.

Discussion

In general, this function is used to set toggles in VDA Tool and Navigator menus. For example, the Options menu toggles found in VDA Tools and the Configure menu toggles found on the Navigator are set with this routine.

A submenu’s index number is the number following its parent menu index.

Separators are counted as items in the menu, and must be included in the *item_index* specification.

Example

This code gets the stored status of the Buttonbar, Controls area, and Message area and sets the menu toggles to match.

(From wave/lib/vdatools/wzscheight.pro.)
; Set the Options menu toggles.
area_status=[$
  TmGetAttribute(tool_name, 'TM', 'BUTTONBAR_STATUS', Default=1), $
  TmGetAttribute(tool_name, 'TM', 'CONTROLS_STATUS', Default=1), $
  TmGetAttribute(tool_name, 'TM', 'MESSAGE_STATUS', $ $
      Default=1) $
  ]$

WoMenuBarSetToggle, tool_name, [7, 7, 7], [1, 2, 3], area_status

See Also
WoMenuBar

WoMessage Function

Creates a message area for a VDA Tool

Usage

\[ widget = \text{WoMessage}(parent, toolname) \]

Input Parameters

\text{parent} — Specifies the parent widget ID of the message area (long).

\text{toolname} — (string) Specifies the unique name of the VDA Tool to which the message area is attached.

Returned Value

\text{widget} — The ID of the message area widget.

Keywords

\text{Frame} — If present and nonzero, a frame widget is placed around the message area.

\text{Rows} — Specifies the number of lines in the message area.
Attachment Keywords

**Bottom** — If a widget ID is specified (for example, \texttt{Bottom=wid}), then the bottom of the message area is attached to the top of the specified widget. If no widget ID is specified (for example, \texttt{/Bottom}), then the bottom of the message area is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, \texttt{Left=wid}), then the left side of the message area is attached to the right side of the specified widget. If no widget ID is specified (for example, \texttt{/Left}), then the left side of the message area is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, \texttt{Right=wid}), then the right side of the message area is attached to the left side of the specified widget. If no widget ID is specified (for example, \texttt{/Right}), then the right side of the message area is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, \texttt{Top=wid}), then the top of the message area is attached to the bottom of the specified widget. If no widget ID is specified (for example, \texttt{/Top}), then the top of the message area is attached to the top of the parent widget.

Examples

The following examples show three ways in which WoAddMessage can be used in a VDA Tool. Messages are defined in a resource file, and WoMessage is used to create the message area in the VDA Tool.

This is part of a resource file defined for a VDA Tool.

mytool.ads

```vda
MyTool_intro: Welcome to my tool...
MyTool_SelectButton_1: Press MB1 to begin selection
MyTool_SelectButton_2: Press MB2 to end selection
MyTool_FileRead: Reading file:
...
; Create a 6 line message area.
...
ms = WoMessage(layout, tool_name, Rows=6, /Frame, /Left, $ /Right, /Bottom)
...
```

WoMessage Function 567
Example 1

Introductory messages are added to the VDA Tool.

WoAddMessage, tool_name, ‘MyTool_intro’, /Clear
WoAddMessage, tool_name, ‘WzGlobal_intro’

Example 2

WoAddMessage can be used to display multiple messages with a single call.

WoAddMessage, tool_name, ‘MyTool_SelectButton’, /Clear

Example 3

Messages can be tailored with dynamic information and then displayed.

msg = TmGetMessage(‘mytool.ads’, ‘MyTool_FileRead’)  
msg = msg + ‘ ’ + file_name  
WoAddMessage, tool_name, Message=msg

See Also

WoAddMessage

---

**WoSaveAsPixmap Procedure**

Saves graphics from a specified VDA Tool as a pixmap.

Usage

WoSaveAsPixmap, tool_name, varname

Input Parameters

*tool_name* — The unique name of the VDA Tool from which to read the graphics as a pixmap.

*varname* — The name of the variable in which to save the pixmap.
Keywords

Add — If nonzero, the pixmap is added to the specified variable. This keyword is used to build a sequence of pixmaps that can be animated.

Discussion

The Add keyword is used to build a sequence of pixmaps to animate. The variable created when Add is used is 3D. The first two dimensions contain the pixmap data, and the third dimension represents the number of pixmaps in the variable. For example, if the dimension of the variable is 512-by-512-by-30, the variable, when viewed in the WzAnimate Tool, would produce a loop or cycle consisting of 30 frames.

Example

WzSurface, DIST(30)
   ; Create a surface plot.
WoSaveAsPixmap, ‘WzSurface_0’, ‘Surf’
   ; Save a pixmap of the surface in a variable called “Surf”.
   ; Now, rotate the Surface in the WzSurface Tool.
WoSaveAsPixmap, ‘WzSurface_0’, ‘Surf’, /Add
   ; Add the rotated surface the pixmap variable “Surf”.
WzAnimate, surf
   ; Run the WzAnimate Tool and animate the pixmaps.

WoSetCursor Procedure

Changes the cursor for a VDA Tool.

Usage

WoSetCursor, tool_name

Parameters

tool_name — A string containing the unique name of a VDA Tool.
Keywords

Wait — Displays a “wait” cursor.
System — Displays the system cursor. This is the default.

Discussion

This routine calls WtCursor to change the current cursor.

Example

The following line changes the cursor in the VDA Tool WzPlot_0 to a “wait” cursor.
WoSetCursor, "WzPlot_0", /Wait

See Also

WtCursor

WoSetToolIcon Procedure

Assigns a pixmap to be the icon for a VDA Tool.

Usage

WoSetToolIcon, tool_name, icon

Input Parameters

tool_name — The name of the VDA Tool for which to set the icon.
icon — The filename of the icon pixmap.

Keywords

Appdir — A string that specifies the application directory name. This is the directory in which the application searches for resource files, string resource files, and icon files. See the Discussion. (Default: vdatools)
Discussion

When a VDA Tool is iconized (the user clicks on the appropriate Window Manager button), the specified pixmap is shown on the icon.

**UNIX USERS**  By default, the function looks for *file* first in directories specified by the environment variable `WAVE_RESPATH`. This environment variable is a colon separated list of directories, similar to the PV-WAVE `WAVE_PATH` environment variable. If not found in a `WAVE_RESPATH` directory, the directory `$WAVE_DIR/xres/!Lang/vdatools` is searched, where `!Lang` represents the value of the `!Lang` system variable in PV-WAVE.

**OpenVMS USERS**  By default, the function looks for *file* first in directories specified by the logical `WAVE_RESPATH`. This logical is a comma separated list of directories and text libraries, similar to the OpenVMS `WAVE_PATH` logical. If not found in a `WAVE_RESPATH` directory, the directory `WAVE_DIR:[XRES. !Lang.VDATOOLS]` is searched, where `!Lang` represents the value of the `!Lang` system variable in PV-WAVE.

**Windows USERS**  By default, the function looks for *file* first in directories specified by the environment variable `WAVE_RESPATH`. This environment variable is a semicolon separated list of directories, similar to the PV-WAVE `WAVE_PATH` environment variable. If not found in a `WAVE_RESPATH` directory, the directory `%WAVE_DIR%\xres\!Lang\vdatools` is searched, where `!Lang` represents the value of the `!Lang` system variable in PV-WAVE.

If `Appdir` is specified, the application searches for resources in the following directory:

- **(UNIX)**  `< wavedir>/xres/!Lang/appdir`
- **(OpenVMS)**  `< wavedir>:[XRES. !Lang.APPDIR]`
- **(Windows)**  `< wavedir>\xres\!Lang\appdir`

Where `< wavedir>` is the main PV-WAVE directory.

Example

This call is taken from the code for the WzContour VDA Tool. It sets the icon pixmap for WzContour.

...
**WoSetWindowTitle Procedure**

Sets the title for a VDA Tool window.

**Usage**

WoSetWindowTitle, *tool_name*, *window_title* [, *tool_title*]

**Input Parameters**

*tool_name* — A string specifying the unique name of a VDA Tool.

*window_title* — A string, as returned by WoGetUniqueWindowTitle, specifying a unique name for the VDA Tool window.

*tool_title* — (optional) A string specifying a general name for the VDA Tool, e.g., 'surface'.

**Keywords**

None.

**Example**

See the example for WoGetUniqueWindowTitle.

**See Also**

WoGetUniqueWindowTitle
**WoStatus Function**

Create a status bar for a VDA Tool.

**Usage**

\[\text{widget} = \text{WoStatus}(\text{parent, toolname})\]

**Input Parameters**

- **parent** — Specifies the parent widget ID of the status bar (long).
- **toolname** — (string) Specifies the unique name of the VDA Tool to which the status bar is attached.

**Returned Value**

- **widget** — The ID of the status bar widget.

**Keyword**

- **Frame** — If present and nonzero, a frame widget is placed around the status bar.

**Attachment Keywords**

- **Bottom** — If a widget ID is specified (for example, Bottom=\text{wid}), then the bottom of the status bar is attached to the top of the specified widget. If no widget ID is specified (for example, /Bottom), then the bottom of the status bar is attached to the bottom of the parent widget.

- **Left** — If a widget ID is specified (for example, Left=\text{wid}), then the left side of the status bar is attached to the right side of the specified widget. If no widget ID is specified (for example, /Left), then the left side of the status bar is attached to the left side of the parent widget.

- **Right** — If a widget ID is specified (for example, Right=\text{wid}), then the right side of the status bar is attached to the left side of the specified widget. If no widget ID is specified (for example, /Right), then the right side of the status bar is attached to the right side of the parent widget.

- **Top** — If a widget ID is specified (for example, Top=\text{wid}), then the top of the status bar is attached to the bottom of the specified widget. If no widget ID is specified
(for example, /Top), then the top of the status bar is attached to the top of the parent widget.

**Examples**

The following examples show two ways in which WoAddStatus can be used in a VDA Tool. Messages are defined in a resource file, and WoStatus is used to create the status area in the VDA Tool.

This is part of a resource file defined for a VDA Tool.

```
mytool.ads

  MyTool_Initialize: Initializing...

  MyTool_FileRead:   Read file:

...

; Create the status area.
...

sa = WoStatus(layout, tool_name, Top=bar, /Left)
...
```

**Example 1**

Set the status string.

```
WoAddStatus, tool_name, 'MyTool_Initialize'
```

**Example 2**

Status messages can be tailored with dynamic information and then displayed.

```
msg = TmGetMessage('mytool.ads', 'MyTool_FileRead')
msg = msg + file_name
WoAddStatus, tool_name, Status=msg
```

**See Also**

`WoAddStatus`, `WoMessage`
WoVariableOptionMenu Function

Creates an option menu containing the names of all of the variables associated with the current tool.

**Usage**

\[ widget = \text{WoVariableOptionMenu}(\text{parent}, \text{toolname}) \]

**Input Parameters**

- `parent` — Specifies the parent widget ID of the option menu (long).
- `toolname` — (string) Specifies the unique name of the VDA Tool to which the option menu is to be attached.

**Returned Value**

- `widget` — The widget ID of the option menu.

**Keywords**

- `Start_Value` — Specifies a string containing the name of the initially selected variable.
- `Variables` — A 1D string array containing the names of variables to list in the option menu. By default, the list of variables returned by TmEnumerateVars is used.

**Attachment Keywords**

- `Bottom` — If a widget ID is specified (for example, `Bottom=wid`), then the bottom of the option menu is attached to the top of the specified widget. If no widget ID is specified (for example, `/Bottom`), then the bottom of the option menu is attached to the bottom of the parent widget.
- `Left` — If a widget ID is specified (for example, `Left=wid`), then the left side of the option menu is attached to the right side of the specified widget. If no widget ID is specified (for example, `/Left`), then the left side of the option menu is attached to the left side of the parent widget.
- `Right` — If a widget ID is specified (for example, `Right=wid`), then the right side of the option menu is attached to the left side of the specified widget. If no widget ID is specified (for example, `/Right`), then the right side of the option menu is attached to the right side of the parent widget.
ID is specified (for example, /Right), then the right side of the option menu is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, Top=wid), then the top of the option menu is attached to the bottom of the specified widget. If no widget ID is specified (for example, /Top), then the top of the option menu is attached to the top of the parent widget.

**Discussion**

The first option menu item is labelled *None*.

The variable name option menu cannot be modified once it has been created, except to set the current selection. If the list of variable names changes, the option menu must be recreated.

---

**NOTE** Do not use WwSetValue with the Userdata keyword on the returned widget or on its parent widget after calling this routine. This use will cause unexpected side-effects, because WwSetValue changes the user data for a widget and for all of its children.

---

**Example**

This example creates a variable option menu inside a layout widget. The menu contains the names of the variables associated with the VDA Tool to which the menu is attached.

```
PRO OkApplyCB, wid, which
    ; ...
    ; Retrieve the currently selected variable from the option menu. The returned value will be a string containing the variable name.
    VariableWid = WwGetValue (wid, /Userdata)
    variable = WoVariableOptionMenuGetValue (VariableWid)
    ; ...
END

PRO ChangeVariableCB, wid, which
    ; ...
    ; Set the currently selected variable name in the variable option menu.
    ; The new variable name should be a string containing the name of an existing WAVE variable.
    VariableWid = WwGetValue (wid, /Userdata)
```
WoVariableOptionMenuGetValue Function

Gets the currently selected variable name from an option menu that was created with the WoVariableOptionMenu function.

Usage

\[ \text{varname} = \text{WoVariableOptionMenuGetValue}(\text{wid}) \]

Input Parameters

\text{wid} — Specifies the widget ID returned by WoVariableOptionMenu (long).

Returned Value

\text{varname} — A string containing the name of the currently selected variable.
**Keywords**

None.

**Example**

See the example given for WoVariableOptionMenu.

**See Also**

WoVariableOptionMenu, WoVariableOptionMenuSetValue

---

**WoVariableOptionMenuSetValue Procedure**

Sets the current selection in the variable option menu.

**Usage**

WoVariableOptionMenuSetValue, wid, value

**Input Parameters**

*wid* — Specifies the widget ID returned by WoVariableOptionMenu (long).

*value* — Specifies a string containing the new variable name to set.

**Keywords**

None.

**Discussion**

The variable name option menu cannot be modified once it has been created, except to set the current selection. If the list of variable names changes, the option menu must be recreated.

If no matching option menu values are found, the label None is used.

**Example**

See the example given for WoVariableOptionMenu.
See Also

WoVariableOptionMenu, WoVariableOptionMenuGetValue
Localizing PV-WAVE Applications

To localize an application means to translate the text that appears in the application to the user’s native language. This text includes labels that appear on menus, dialogs, and other GUI “widgets”, as well as error and warning messages.

Localizing the VDA Tools

PV-WAVE uses resource and string files to isolate the dynamic aspects of applications in files that are distinct from the code. For instance, the labels that appear in VDA Tools menus are maintained in separate resource files, and the application accesses these resource files as they are needed. Similarly, error messages and other dynamic text information not directly related to widgets are maintained in separate string files. String and resource files greatly simplify the process of localizing software applications.

The process of translating, or localizing, an application is described in the following sections.

Copy and Translate Resource and String Files

Step 1  Make a copy of the following directory, including its contents and all of its subdirectories:

(UNIX)  <wavedir>/xres/american/*
(OpenVMS)  <wavedir>:XRES.AMERICAN.*
(Windows)  <wavedir>xres\american\*
Where <wavedir> is the main PV-WAVE directory.

**Step 2**  Give the new local-language directory tree a name that reflects the local language. For example: `japanese`.

**Step 3**  Translate the labels and text strings contained in the resource and string files in the new local-language resource directory tree. By convention, string files end in `.ads`, and resource files end in `.ad`.

**NOTE**  Be sure that you can select a font that contains the special characters you wish to use. For special characters to be displayed correctly, a font containing those characters must be selected.

---

**How PV-WAVE Determines the Locale**

PV-WAVE determines which locale (wave/xres subdirectory) to use when loading resource and string files at run time. It does this by examining the value of the !Lang system variable.

To set the initial value of !Lang (the locale), PV-WAVE follows this procedure:

**Step 1**  Looks at the value of the `WAVE_LANG` environment variable (or VMS logical).

**Step 2**  If step 1 fails, looks at the value of the locale that is defined by the operating system.

**Step 3**  If steps 1 and 2 fail, uses a default locale.

The rest of this section explains these steps in detail. Using the information in this section, you can set the default locale of your entire PV-WAVE installation, or override the default locale for a particular PV-WAVE session.

**The WAVE_LANG Environment Variable**

To determine the locale to use, PV-WAVE first checks the environment variable `WAVE_LANG`.

When PV-WAVE starts, the environment variable (or VMS logical) `WAVE_LANG` is examined. If this environment variable is set and if the directory:

- (UNIX)  `<wavedir>/xres/$WAVE_LANG`
- (OpenVMS)  `<wavedir>:[XRES.WAVE_LANG]`
- (Windows)  `<wavedir>\xres\%WAVE_LANG%`
exists, then WAVE_LANG is used to set the !Lang system variable and, thus, the system locale. For instance, if WAVE_LANG is set to "japanese", then the resource and string files for PV-WAVE applications will be loaded from the area:

(UNIX)  <wavedir>/xres/japanese
(OpenVMS)  <wavedir>:[XRES.JAPANESE]
(Windows)  <wavedir>xres\JAPANESE

**NOTE** Use WAVE_LANG primarily to override the default system locale. For example, if the default system locale is "japanese", you can use the American English resources by setting WAVE_LANG to "american".

### The Operating System Locale Setting

If the WAVE_LANG variable is not set, or if its value cannot be used (for example, because a subdirectory named $WAVE_DIR/xres/$WAVE_LANG does not exist), then the operating system locale setting is retrieved and used to determine the proper locale for PV-WAVE.

The operating system locale setting is determined by the value of an environment variable (or VMS logical). If necessary, PV-WAVE examines this variable and uses its value as a key into a string file called locale.ads, which resides in the xres directory.

The locale.ads file contains name/value pairs, with the name of the operating system locale on the left, followed by a colon, followed by the name of the xres subdirectory containing the localized resource files. For example:

Japanese_Japan.932: japanese

could be used to specify Japanese as the locale. In this case, if the operating system locale is Japanese_Japan.932 (the standard Japanese locale value for Windows systems), then PV-WAVE will set its default locale to "japanese".

**UNIX and OpenVMS USERS** You can check the default locale setting for your operating system by entering one of the following commands at your operating system prompt:

(UNIX)  echo $LANG
(OpenVMS)  SHOW LOGICAL SYS$LANGUAGE
Windows USERS  You can check the default locale setting for your operating system by entering the following command at the WAVE> prompt:

```
WAVE> PRINT, WIN32_LOCALE()
```

**Default Locale Setting**

If PV-WAVE is unable to set the locale using the WAVE_LANG environment variable or the locale.ads file, then the default locale american is used. You can change this default for the entire PV-WAVE installation by modifying the following line in the locale.ads file:

```
DEFAULT: american
```

For example, to change the default locale for your PV-WAVE installation to japanese, change this line to:

```
DEFAULT: japanese
```

---

**Localizing the PV-WAVE Home Window**

**NOTE**  The Home Window is only used in the Microsoft Windows versions of PV-WAVE.

**View the Resources**

**Step 1**  Open the Microsoft Visual C++ Developer Studio version 5.0.

**NOTE**  Other MSVC versions or other resource editors may work but have not been tested.

**Step 2**  From the File menu, select Open. From the Open dialog box, locate the Open As option, then select Resources from the option list.

**Step 3**  Open the file wavewin2.exe, located in

- Windows NT or Windows 95:
  `%WAVE_DIR%\bin\bin.i386nt`
• Alpha NT:
  %WAVE_DIR%\bin\bin.ALPHAnt

The resources for wavewin2.exe now appear in a window.

**Localize Your Resources**

After the PV-WAVE Home Window resources are displayed, you can edit them to suit your particular locale.

This section presents an example procedure for modifying a PV-WAVE Home Window resource. Copies of all resources are made prior to localizing them to preserve the English language resources.

**Step 1**  Double-click on the wavewin2.exe folder so all its subfolders are displayed.

**Step 2**  Select a resource to modify. In this following example, we will modify the Menu resources.

**Step 3**  Double-click on the Menu folder so its resource ID displays (in this case, 128).

**Step 4**  Click the right mouse button on the resource ID (in this case, 128) and select Insert Copy from the pop-up menu.

**Step 5**  Select the name of your locale from the list of languages and click the OK button.

**Step 6**  Double-click on the newly created menu’s resource ID. The PV-WAVE Home Windows menu bar will appear.

**Step 7**  Double-click on a menu bar resource, for example, the File menu. The Menu Item Properties dialog will display.

**Step 8**  Modify the text in the Caption field.

**Step 9**  Save the new executable when you are finished modifying the resources. The modified resource(s) will be used next time the PV-WAVE Home Window is started.
Localizing the PV-WAVE Print Dialog Box

NOTE The Print dialog is only used in the Microsoft Windows versions of PV-WAVE when the WIN32_PICK_PRINTER command is run. For more information on this command, see the description of the Windows Metafile (WMF) Driver in Appendix B of the PV-WAVE Reference.

View the Resources

Step 1 Open the Microsoft Visual C++ Developer Studio version 5.0.

NOTE Other MSVC versions or other resource editors may work but have not been tested.

Step 2 From the File menu, select Open. From the Open dialog box, locate the Open As option, then select Resources from the option list.

Step 3 Open the file vnigraph.dll, located in

- Windows NT or Windows 95:
  %WAVE_DIR%\bin\bin.i386nt
- Alpha NT:
  %WAVE_DIR%\bin\bin.ALPHAnt

The resources for vnigraph.dll now appear in a window.

Localize Your Resources

After the PV-WAVE Print dialog resources are displayed, you can edit them to suit your particular locale.

This section presents an example procedure for modifying a PV-WAVE Print dialog resource. Copies of all resources are made prior to localizing them to preserve the English language resources.

Step 1 Double-click on the vnigraph.dll folder so all its subfolders are displayed.

Step 2 Double-click on the Dialog folder so its resource ID displays (in this case, 1538).
Step 3  Click the right mouse button on the resource ID (in this case, 1538) and select **Insert Copy** from the pop-up menu.

Step 4  Select the name of your locale from the list of languages and click the **OK** button.

Step 5  Double-click on the newly created dialog’s resource ID. The PV-WAVE Print dialog will appear.

Step 6  Double-click on a dialog resource, for example, the **Name** label. The **Text Properties** dialog will display.

Step 7  Modify the text in the **Caption** field.

Step 8  Save the new executable when you are finished modifying the resources. The modified resource(s) will be used next time the PV-WAVE Print dialog is selected.
Motif Widget Classes

This appendix lists the Motif widget classes and convenience widgets.

Motif Widget Classes

The following widget classes are defined in the file

```
wxmclasses.pro
```

in the Standard Library. They are used as a parameter to the WtCreate function. For example:

```
widget=WtCreate('Done', xmPushButtonWidgetClass, top)
```

**NOTE** Widget classes are only used with Widget Toolbox functions. WAVE Widgets users do not need to use this appendix.

overrideShellWidgetClass
vendorShellWidgetClass
transientShellWidgetClass
topLevelShellWidgetClass
applicationShellWidgetClass
xbaeMatrixWidgetClass
xmArrowButtonWidgetClass
xmBulletinBoardWidgetClass
xmCascadeButtonWidgetClass
xmCommandWidgetClass
xmDialogShellWidgetClass
xmDrawingAreaWidgetClass
xmDrawnButtonWidgetClass
xmFileSelectionBoxWidgetClass
xmFormWidgetClass
xmFrameWidgetClass
xmLabelWidgetClass
xmListWidgetClass
xmMainWindowWidgetClass
xmMenuShellWidgetClass
xmMessageBoxWidgetClass
xmPanedWindowWidgetClass
xmPushButtonWidgetClass
xmRowColumnWidgetClass
xmScaleWidgetClass
xmScrollBarWidgetClass
xmScrolledWindowWidgetClass
xmSelectionBoxWidgetClass
xmSeparatorWidgetClass
xmTextWidgetClass
xmTextFieldWidgetClass
xmToggleButtonWidgetClass
xmArrowButtonGadgetClass
xmCascadeButtonGadgetClass
Convenience Widgets

Convenience widgets allow easy manipulation of predefined groups of widgets. Convenience widgets are unmanaged when created. To make them visible use WtGet(..., /Manage) to manage them.

For more information on convenience widgets, see the OSF/Motif Programmer’s Guide.

NOTE  The convenience widgets listed here are not related to WAVE Widgets.

BulletinBoardDialogWidget
ErrorDialogWidget
FileSelectionDialogWidget
FormDialogWidget
InformationDialogWidget
MenuBarWidget
MessageDialogWidget
OptionsMenuWidget
PopupMenuWidget
PromptDialogWidget
PulldownMenuWidget
QuestionDialogWidget
RadioBoxWidget
ScrolledListWidget
ScrolledTextWidget
SelectionDialogWidget
SimpleCheckBoxWidget
SimpleMenuBarWidget
SimpleOptionMenuWidget
SimplePopupMenuWidget
SimplePulldownMenuWidget
SimpleRadioBoxWidget
WarningDialogWidget
WorkAreaWidget
WorkingDialogWidget
Motif Callback Parameters

This appendix describes the parameters that are required in callback routines for Widget Toolbox functions under Motif.

For any Widget Toolbox callback under Motif, the first five parameters are always the same; however, some widget classes require additional callback parameters. These required and additional parameters are described below.

**NOTE** The callback parameters described below are only used with Widget Toolbox functions. WAVE Widgets users do not need to use this appendix.

**Required Callback Parameters**

The callback routines for all Motif widgets must have the following five parameters. The only exception to this is when the *Noparams* keyword is used in the WtCreate function. When this keyword is used, callbacks need only two parameters, the *widget* and *data*. All callback parameters are read-only.

- *widget* — The widget ID.
- *data* — Client data passed to WtAddCallback.
- *nparams* — The number of callback parameters after *nparams*. Two are required: *reason* and *event*. 


- **reason** — Callback reason (For example: Xm_CR...).

See the *OSF/Motif Programmer’s Reference* for more information on callback parameters.

### Additional Required Callback Parameters

This section lists additional required callback parameters for the Motif widget classes.

**NOTE** Callbacks for the XbaeMatrix (table) widget and XvnPreview widget are discussed in supplemental XbaeMatrix and XvnPreview documentation, which you can find in the following files:

```
<wavedir>/docs/widgets/matrix_motif.ps
<wavedir>/docs/widgets/preview_motif.ps
```

Where `<wavedir>` is the main PV-WAVE directory.

You can print these files on any PostScript printer.

---

**ArrowButton**

- **click_count** — Number of clicks. Only specified if the callback reason is XmCR_ACTIVATE. See the *OSF/Motif Programmer’s Reference* for more information.

**Command**

- **command** — A string containing the last command entered.
- **length** — Length of the command.

**DrawingArea**

- **window** — Window ID of the drawing area.

**DrawnButton**

- **window** — Window ID of the drawn button.
• **click_count** — Number of clicks. Only specified if the callback reason is XmCR_ACTIVATE. See the *OSF/Motif Programmer’s Reference* for more information.

**FileSelection**

• **filespec** — A string containing the file specification.
• **filespeclen** — File specification length.
• **mask** — A string containing the directory/file mask.
• **masklen** — The directory/file mask length.
• **dir** — A string containing the directory.
• **dirlen** — Directory length.
• **pattern** — A string containing the file search pattern.
• **patternlen** — The file search pattern length.

**List**

• **item** — A string containing the last selected item.
• **itemlen** — The last selected item length.
• **position** — Position of the last selected item (for callback reasons XmCR_MULTIPLE_SELECT and XmCR_EXTENDED_SELECT).
• **selected_items** — An array of strings containing selected items.
• **positions** — An array of long integers specifying positions of the selected items.
• **count** — Number of selected items.
• **selectiontype** — Type of selection (XmINITIAL, XmMODIFICATION, XmADDITION).

**PushButton**

• **click_count** — Number of clicks. Only specified if the callback reason is XmCR_ACTIVATE. See the *OSF/Motif Programmer’s Reference* for more information.

**RowColumn**

The following parameters are only used only if the callback reason is XmCR_ACTIVATE.
- **widget** — Widget ID of the selected RowColumn item.
- **button** — Button number of the selected button.
- **click_count** — Number of clicks. For more information, see the OSF/Motif Programmer’s Reference.

**Scale**
- **value** — New slider value.

**ScrollBar**
- **value** — New slider value.
- **pixel** — X-coordinate of the mouse button for horizontal direction; y-coordinate of the mouse button for vertical direction. Used only for the XmCR_TO_BOTTOM and Xm_CR_TO_TOP callback reasons.

**Selection**
- **selection** — A string containing the selection.
- **selectionlen** — Selected string length.

**Text, TextField**
For the callback reasons:

```
XmCR_LOSING FOCUS,
XmCR MODIFYING _TEXT _VALUE,
XmCR MOVING _INSERT _CURSOR
```
- **doit** — Return value indicating if action is performed — must be declared as a long (e.g.: doit = OL).
- **currInsert** — Current position of insert cursor.
- **newInsert** — New position of insert cursor.
For the callback reasons:

```
XmCR MODIFYING _TEXT _VALUE,
XmCR MOVING _INSERT _CURSOR
```
- **startPos** — Starting position of the text to modify.
- **endPos** — Ending position of the text to modify.
For the callback reason:
XmCR_MODIFYING_TEXT_VALUE

- **text** — A string containing the text to be inserted.
- **textlen** — Text length.
- **format** — Format of the text.

ToggleButton

- **value** — The toggle button’s current state.
Widget Toolbox Cursors

This appendix lists the standard and custom cursors that are available for use with the WtCursor function.

Standard X Cursors

For detailed information on the following cursors, see Appendix I of the *Xlib Reference Manual, Volume 2*, O’Reilly & Associates, Inc.

Windows USERS Only the cursors preceded by * are available on Windows platforms.

- XC_X_cursor
- * XC_arrow
- XC_based_arrow_down
- XC_based_arrow_up
- XC_boat
- XC_bogosity
- * XC_bottom_left_corner
- * XC_bottom_right_corner
- * XC_bottom_side
XC_bottom_tee
XC_box_spiral
XC_center_ptr
XC_circle
XC_clock
XC_coffee_mug
XC_cross
XC_cross_reverse
* XC_crosshair
XC_diamond_cross
XC_dot
XC_dotbox
XC_double_arrow
XC_draft_large
XC_draft_small
XC_draped_box
XC_exchange
* XC_fleur
XC_gobbler
XC_gumby
XC_hand1
XC_hand2
XC_heart
XC_icon
XC_iron_cross
* XC_left_ptr
* XC_left_side
XC_left_tee
XC_leftbutton
XC_ll_angle
XC_lr_angle
XC_man
XC_middlebutton
XC_mouse
XC_pencil
XC_pirate
XC_plus
XC_question_arrow
XC_right_ptr
* XC_right_side
XC_right_tee
XC_rightbutton
XC_rtl_logo
XC_sailboat
XC_sb_down_arrow
XC_sb_h_double_arrow
XC_sb_left_arrow
XC_sb_right_arrow
XC_sb_up_arrow
XC_sb_v_double_arrow
XC_shuttle
XC_sizing
XC_spider
XC_spraycan
XC_star
XC_target
XC_tcross
* XC_top_left_arrow
Custom Cursors

The following custom cursors were developed by Visual Numerics and are available for use with WtCursor:

WIDGET_WAIT_CURSOR
WIDGET_DEFAULT_CURSOR
WIDGET_CROSSHAIR_CURSOR
WIDGET_X_CURSOR
WIDGET_ARROW_CURSOR
WIDGET_VCOLORBAR_CURSOR
WIDGET_HCOLORBAR_CURSOR
WIDGET_MACRO_PLAY_CURSOR
WIDGET_MACRO_RECORD_CURSOR
WIDGET_MAGNIFIER_CURSOR
WIDGET_MACRO_PLAY_CURSOR
WIDGET_MACRO_RECORD_CURSOR
WIDGET_MAGNIFIER_CURSOR
WIDGET_PROF_COL_CURSOR
WIDGET_PROF_END_CURSOR
WIDGET_PROF_ROW_CURSOR
WIDGET_VIEWFINDER_CURSOR
WIDGET_CAUTION_CURSOR
Developing Portable Applications

With PV-WAVE version 6.0 and later versions, it is possible to write PV-WAVE GUI applications that are portable between Motif-based and Microsoft Windows-based systems. Some consideration needs to be given to the design of such applications to ensure they are indeed portable between the two systems. While the vast majority of the commands that comprise the PV-WAVE language will run on all platforms, special consideration to the WAVE Widgets commands in particular is required.

Writing Portable WAVE Widgets Applications

When WAVE Widgets applications were designed with previous versions of PV-WAVE, use of WAVE Widgets (Ww) and Widget Toolbox (Wt) routines could be mixed freely to create the desired interface. However, when writing an application targeted for both Microsoft Windows and Motif, use of Widget Toolbox (Wt) commands is not recommended. In particular, Motif constants as defined in the wtxmconsts.pro and wtxmclasses.pro files have their Microsoft Windows counterparts in the files wtwacconsts.pro and wtwacclasses.pro. These differences as well as others are folded into the Ww layer routines to provide a common Application Programming Interface (API). Hence, to create a portable application, the general rule is to use the Ww layer routines to access Wt layer functionality.

Every effort has been made to encapsulate as much of the common functionality as is practical in the Ww layer routines.
Commands and Filenames

One of the major considerations that affects applications written to run on multiple windowing systems is the differing system commands and file naming conventions imposed by the underlying operating systems. Several PV-WAVE commands are available to help eliminate these differences by constructing pathnames given a filename. These commands are listed in the following table.

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEL_FILE</td>
<td>Deletes a file.</td>
</tr>
<tr>
<td>FINDFILE</td>
<td>Searches for files given a file specification.</td>
</tr>
<tr>
<td>FILEPATH</td>
<td>Constructs pathnames to PV-WAVE files.</td>
</tr>
</tbody>
</table>

In addition, standard PV-WAVE directories can be referenced by using system variables to avoid having to specify platform-specific filenames. In particular the following system variables can be used to specify files in PV-WAVE directories:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!Dir</td>
<td>The top level PV-WAVE directory.</td>
</tr>
<tr>
<td>!Data_Dir</td>
<td>The PV-WAVE data directory.</td>
</tr>
</tbody>
</table>

In cases where the above commands cannot be used to construct the filename, it is important to follow the conventions of the operating systems, as described in the following table.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Format</th>
<th>Case Sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX</td>
<td>/dir1/dir2/.../file</td>
<td>Yes</td>
</tr>
<tr>
<td>OpenVMS</td>
<td>DISK:[DIR1.DIR2...]FILE.EXT</td>
<td>No</td>
</tr>
<tr>
<td>Microsoft Windows</td>
<td>Drive:\dir1\dir2...\file.ext</td>
<td>No</td>
</tr>
</tbody>
</table>

It is also important to remember that since the colon character (:) is used as a valid part of filenames on OpenVMS and Microsoft Windows, a colon should not be used to separate directories in a path list if one is required. By convention, the comma character (,) is used to separate pathnames on OpenVMS systems and the
semicolon (;) on Microsoft Windows systems. Since UNIX filenames may not contain the colon character, it is used to separate directories on UNIX platforms.

**WwGetKey Routine**

Due to differences in the underlying implementation of event/message processing, differences exist in the processing of key events between Motif and Microsoft Windows. The WwGetKey routine bridges the majority of these differences. However, when using the *Keysym* keyword with WwGetKey, differences in values may be seen, especially in the translation of unprintable keys like <Delete> or <Return>.

**Specifying WAVE Widget Fonts**

On Motif-based systems, it is possible to customize WAVE Widgets that contain text by changing the font in which the text is displayed. This is often done by specifying the *Font* keyword in the WAVE Widget (Ww) routine call. It is also possible to change the font through a call to WwSetValue or by specifying the font in a resource file.

The Microsoft Windows version of PV-WAVE 6.0 does not provide support for changing widget fonts. Widgets that display text will do so using the default Microsoft Windows system font. One notable exception is the Preview widget. Because of the column-oriented nature of the widget, it cannot be used effectively with a proportional font and therefore displays in the default fixed width font.

**TIP** All requests to change a widget font will be accepted without issuing an error, but will not alter the displayed font.

**Display Sizes**

Keep in mind when writing GUI applications for Microsoft Windows that some systems may be running at lower resolutions. It is not uncommon for Microsoft Windows to be configured at 800x600, or 640x480, which is significantly lower than the standard resolution of most workstation displays. This is important when, for instance, drawn buttons are displayed. Since bitmaps are used as the image in the button, the size cannot be reduced without altering the bitmap. The PV-WAVE Navigator application has a *Lowres* keyword that reconfigures the button bar to display two rows of buttons to account for lower resolution displays. The *Lowres* keyword also reduces the size of the drawing area in graphics VDA Tools to better fit the display.
Windows Graphics

Both OpenVMS and UNIX platforms use the X Windows graphics driver and thus behave identically. The Win32 driver, which is used to provide PV-WAVE graphics on Microsoft Windows-based systems, was written to be compatible with the X Windows driver in as many ways as possible; however, several notable differences exist between the two.

Graphics Cache

To be more efficient, the Win32 graphics driver now uses buffered output. This leads to rare occasions where a program needs to be certain that data are not waiting in a buffer, but have actually been output. The most common side-effect of buffered graphics is seen in improperly stacked graphics. The EMPTY procedure can be used to flush all buffered output.

EMPTY is a low-level graphics routine. PV-WAVE graphics routines generally handle the flushing of buffered data transparently, so the need for EMPTY is rare. Graphics VDA Tools writers will need to be more aware of buffered graphics to ensure proper stacking of the Graphical Elements with respect to the tool’s graphics.

When executed on a device that does not use buffered output, the EMPTY command has no effect.

Color Tables

The handling of color tables differs somewhat between the X Windows and the Microsoft Windows platforms. For the sake of this discussion, the assumption is that X Windows is running in 8-bit pseudo-color mode and Microsoft Windows is running in 256-color mode, since these modes are where differences are most apparent.

In X Windows, colormap entries are allocated, if available, in a shared colormap. This colormap is shared between all the applications running on the workstation and is limited to 256 entries. In PV-WAVE, entries in this shared colormap are reserved when the first graphics window is created. It is possible at the time of this allocation to limit the number of colormap entries reserved by specifying the Colors keyword to the WINDOW command.

The value specified with the Colors keyword can take two forms: 1) A positive value \( n \), specifying “allocate \( n \) entries” or 2) A negative value \( -n \), specifying “allocate all but \( n \) entries.” The value of the PV-WAVE system variable \!D.N_Colors\ reports the actual number of color indices available.
Microsoft Windows differs from X Windows in several respects. First, colormaps are not specifically shared between applications. The currently active application, known as the “foreground” application, has its colormap installed and its colors are displayed correctly. All other applications are given the foreground application’s colormap and attempt to display their data as best they can. Secondly, the Win32 driver has access to a 256-color palette. Of these colors, 20 are reserved for Microsoft Windows GUI elements, leaving 236 colors. For efficiency and compatibility reasons the driver indicates that it has 256 colors and the variable !D.N_Colors reports 256. In actuality, 20 of those colors are simply mappings into the closest available color of the other 236 colors.

On Microsoft Windows, the *Colors* keyword to the WINDOW command has no effect.

**Graphics Function**

The DEVICE command provides support for setting alternative graphics functions through the *Set_Graphics_Function* keyword. The graphics function is a logical function that specifies how the source pixel values generated by a graphics operation are combined with the pixel values already present on the screen.

Graphics functions operate on pixel values. Hence, if you are combining pixel values 7 (00000111) and 21 (00010101) with the XOR operation, the resulting pixel value would be 18 (00010010). The actual color on the screen will be dependent on the installed colormap.

Since the PV•WAVE X Windows driver allocates colors in a shared colormap, the color values specified in PV•WAVE commands do not translate directly into X Windows pixel values. For example, the command

\texttt{PLOT, y, Color=0}

specifies color 0 for the plotted line. This will not likely result in pixel value 0 being used, since 0 is typically reserved by the window manager. The *Color* keyword’s value is translated into a pixel value and the graphics function is applied to this value.

On Microsoft Windows-based systems, since colormaps are not shared among applications, color values translate directly into pixel values and the results may differ greatly from X Windows. One notable example is in using the XOR graphics function with the *Color* keyword value as 0. On X Windows, as explained above, the pixel value used with the XOR function will likely not be 0 and will be combined with the destination pixels and drawn in a third pixel value. On Microsoft Windows, the color value 0 maps into the pixel value 0. Thus, when combined with the destination pixel values, according to the XOR function’s rules, the resulting
pixel values will be exactly the same as the destination pixels and the graphics will appear unchanged.

**WINDOW Keyword Differences**

Several keywords to the WINDOW command that are specific to the X Windows driver have been altered to better fit the Microsoft Windows parlance. The following table lists X Windows keywords and the Microsoft Windows counterparts.

<table>
<thead>
<tr>
<th>X Windows</th>
<th>Microsoft Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get_XWin_ID</td>
<td>Get_Win_ID</td>
</tr>
<tr>
<td>Set_XWin_ID</td>
<td>Set_Win_ID</td>
</tr>
<tr>
<td>Pixmap</td>
<td>Bitmap</td>
</tr>
</tbody>
</table>

**NOTE** For compatibility reasons, the X Windows WINDOW command keywords will also work on Microsoft Windows.

---

**Writing Portable VDA Tools**

In addition to the topics covered previously on writing portable WAVE Widgets applications, several special considerations need to be made when developing VDA Tools for use on Microsoft Windows and Motif platforms.

**VDA Utilities and Tools Manager Routines**

First, the VDA Utilities (Wo) and Tools Manager (Tm) routines have been specifically designed to work on all PV-WAVE platforms and their use is highly encouraged. In situations where differences exist between systems, for instance in file-naming conventions, VDA Utilities routines help bridge the differences.

**VDA Tools Resource Files**

VDA Tools make extensive use of resource files for internationalization and customization. These resource files are portable between the Motif and Microsoft Windows systems. However, a few notable differences exist. First, the VDA Tools’
The appropriate global resource file is different between the various systems. The following table defines this usage:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Global Resource File</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX</td>
<td>$WAVE_DIR/xres/american/vdatools/VDATools</td>
</tr>
<tr>
<td>OpenVMS</td>
<td>WAVE_DIR:[XRES.AMERICAN.VDATOOLS]VDATOOLS.DAT</td>
</tr>
<tr>
<td>Windows</td>
<td>%WAVE_DIR%/xres/american/vdatools/windows/wzglobal.ad</td>
</tr>
</tbody>
</table>

The appropriate global resource file is loaded automatically when the VDA Tools environment is initialized, so knowledge of this difference is only important when changes or additions are made to the global resources.

Secondly, the bitmap format used for drawn buttons and icons is different between the Motif and Microsoft Windows systems. Motif uses the xbm (X Bitmap) format to specify bitmaps and the pm (Pixmap) format for pixmaps. Microsoft Windows uses the BMP format for both. The VDA Utility routine WoBuildResourceFilename can be used to construct the correct path given the filename. 

WoBuildResourceFilename builds a path internally, which it then searches to locate the files. For Microsoft Windows platforms, the vdatools\windows directory will be searched prior to the vdatools directory. Exploiting this feature, the following code segment from the file $WAVE_DIR/lib/vdatools/wographicsbuttons.pro shows one method of specifying drawn buttons by using the WoBuildResourceFilename routine:

```plaintext
IF STRMATCH(!Version.OS, '[Ww][Ii][Nn]') $
THEN ext = '.bmp' $
ELSE ext = '.xbm'

graphics_buttons_toggle = {, $
  LAYOUT_NAME: 'graphicsButtons_1', $
  DESCRIPTOR: 'redraw', $
  CALLBACK: 'WoGBBRedrawCB', $
  STATUS_CALLBACK: 'WoGBBRedrawStatusCB', $
  INSENSITIVE_PIXMAP: $
    WoBuildResourceFilename$
      ('redrawoperationx'+ext), $
  PIXMAP: WoBuildResourceFilename$
    (&'redrawoperation'+ext), $
  DESCRIPTOR: 'cut', $
  CALLBACK: 'WoGBBCutCB', $
  STATUS_CALLBACK: 'WoGBBCutStatusCB', $
```
INSENSITIVE_PIXMAP: $WoBuildResourceFilename$ ('cutoperationx'+ext), $PIXMAP: WoBuildResourceFilename$ ('cutoperation'+ext), $

NOTE The VDA utilities routine WoSetToolIcon requires a filename without an extension and constructs the correct path name in all systems.

---

**Writing System-specific Code**

In situations where system-specific code is necessary, for instance when specifying a filename, the !Version. OS system variable can be used to determine which system is running the procedure. For instance the following code segment allows different commands to be executed for Microsoft Windows (Windows 95 and Windows NT), UNIX, and OpenVMS systems:

```plaintext
IF STRMATCH(!Version.OS, '\[Ww][Ii][Nn]') $
  THEN BEGIN
    ; Windows code
  ENDIF ELSE IF STRMATCH(!Version.OS, $'
    '\[Vv][Mm][Ss]'$
    ; VMS code
  ENDIF ELSE BEGIN
    ; Unix code
ENDELSE
```

Furthermore, if a more specific distinction needs to be made between operating systems at runtime, use the value of the !Version.OS system variable for your platform as defined in the following table:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Operating System</th>
<th>!Version.OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital Alpha</td>
<td>OpenVMS</td>
<td>vms</td>
</tr>
<tr>
<td>Digital Alpha</td>
<td>Digital UNIX</td>
<td>OSF1</td>
</tr>
</tbody>
</table>
Several good examples of portable code are available in your PV-WAVE (version 6 and later) distribution. One good source is the PV-WAVE Gallery, found in the gallery directory.

The PV-WAVE procedure files defined in this directory are completely portable between Microsoft Windows and Motif.

Another good set of examples of portable code is the VDA Tools. These are written to be portable between Microsoft Windows and Motif using the VDA Utilities and Tools Manager routines to bridge system differences. The following table shows the location of the VDA Tools routines.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Operating System</th>
<th>!Version.OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital VAX</td>
<td>OpenVMS</td>
<td>vms</td>
</tr>
<tr>
<td>HP 9000/s700</td>
<td>HPUX</td>
<td>hp-ux</td>
</tr>
<tr>
<td>IBM RS/6000</td>
<td>AIX</td>
<td>AIX</td>
</tr>
<tr>
<td>Intel 486, Pentium</td>
<td>Microsoft Windows NT</td>
<td>Windows-NT</td>
</tr>
<tr>
<td>Intel 486, Pentium</td>
<td>Microsoft Windows 95</td>
<td>Windows</td>
</tr>
<tr>
<td>Silicon Graphics</td>
<td>IRIX</td>
<td>IRIX</td>
</tr>
<tr>
<td>Sun4/SPARC</td>
<td>Solaris</td>
<td>solaris</td>
</tr>
<tr>
<td>Sun4/SPARC</td>
<td>SunOS</td>
<td>sunos</td>
</tr>
</tbody>
</table>

**Example Code**

Several good examples of portable code are available in your PV-WAVE (version 6 and later) distribution. One good source is the PV-WAVE Gallery, found in the gallery directory.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Gallery Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX</td>
<td>$WAVE_DIR/demo/gallery3</td>
</tr>
<tr>
<td>OpenVMS</td>
<td>WAVE_DIR:[demo.gallery3]</td>
</tr>
<tr>
<td>Microsoft Windows</td>
<td>$WAVE_DIR$\demo\gallery3</td>
</tr>
</tbody>
</table>

The PV-WAVE procedure files defined in this directory are completely portable between Microsoft Windows and Motif.

Another good set of examples of portable code is the VDA Tools. These are written to be portable between Microsoft Windows and Motif using the VDA Utilities and Tools Manager routines to bridge system differences. The following table shows the location of the VDA Tools routines.
For examples of portable non-graphics as well as graphics code, consult the PV-WAVE Standard Library procedures found in:

<table>
<thead>
<tr>
<th>Platform</th>
<th>VDA Tools Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX</td>
<td>$WAVE_DIR/lib/vdatools</td>
</tr>
<tr>
<td>OpenVMS</td>
<td>WAVE_DIR:[lib.vdatools]</td>
</tr>
<tr>
<td>Microsoft Windows</td>
<td>%WAVE_DIR%\lib\vdatools</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Platform</th>
<th>Standard Library Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX</td>
<td>$WAVE_DIR/lib/std</td>
</tr>
<tr>
<td>OpenVMS</td>
<td>WAVE_DIR:[lib.std]</td>
</tr>
<tr>
<td>Microsoft Windows</td>
<td>%WAVE_DIR%\lib\std</td>
</tr>
</tbody>
</table>
# Virtual Keys

The following table shows the symbolic constant names, hexadecimal values, and keyboard equivalents for the virtual-key codes used by the Microsoft Windows operating system. The codes are listed in numeric order.

<table>
<thead>
<tr>
<th>Symbolic Constant Name</th>
<th>Hexadecimal Value</th>
<th>Mouse or Keyboard Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_LBUTTON</td>
<td>01</td>
<td>Left mouse button</td>
</tr>
<tr>
<td>VK_RBUTTON</td>
<td>02</td>
<td>Right mouse button</td>
</tr>
<tr>
<td>VKCANCEL</td>
<td>03</td>
<td>Control-break processing</td>
</tr>
<tr>
<td>VK_MBUTTON</td>
<td>04</td>
<td>Middle mouse button (three-button mouse)</td>
</tr>
<tr>
<td>VK_BACK</td>
<td>08</td>
<td>BACKSPACE key</td>
</tr>
<tr>
<td>VK_TAB</td>
<td>09</td>
<td>TAB key</td>
</tr>
<tr>
<td></td>
<td>0A-0B</td>
<td>Undefined</td>
</tr>
<tr>
<td>VK_CLEAR</td>
<td>0C</td>
<td>CLEAR key</td>
</tr>
<tr>
<td>VK_RETURN</td>
<td>0D</td>
<td>ENTER key</td>
</tr>
<tr>
<td></td>
<td>0E-0F</td>
<td>Undefined</td>
</tr>
<tr>
<td>VK_SHIFT</td>
<td>10</td>
<td>SHIFT key</td>
</tr>
<tr>
<td>Symbolic Constant Name</td>
<td>Hexadecimal Value</td>
<td>Mouse or Keyboard Equivalent</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>VK_CONTROL</td>
<td>11</td>
<td>CTRL key</td>
</tr>
<tr>
<td>VK_MENU</td>
<td>12</td>
<td>ALT key</td>
</tr>
<tr>
<td>VK_PAUSE</td>
<td>13</td>
<td>PAUSE key</td>
</tr>
<tr>
<td>VK_CAPITAL</td>
<td>14</td>
<td>CAPS LOCK key</td>
</tr>
<tr>
<td></td>
<td>15-19</td>
<td>Reserved for Kanji systems</td>
</tr>
<tr>
<td></td>
<td>1A</td>
<td>Undefined</td>
</tr>
<tr>
<td>VK_ESCAPE</td>
<td>1B</td>
<td>ESC key</td>
</tr>
<tr>
<td></td>
<td>1C-1F</td>
<td>Reserved for Kanji systems</td>
</tr>
<tr>
<td>VK_SPACE</td>
<td>20</td>
<td>SPACEBAR</td>
</tr>
<tr>
<td>VK_PRIOR</td>
<td>21</td>
<td>PAGE UP key</td>
</tr>
<tr>
<td>VK_NEXT</td>
<td>22</td>
<td>PAGE DOWN key</td>
</tr>
<tr>
<td>VK_END</td>
<td>23</td>
<td>END key</td>
</tr>
<tr>
<td>VK_HOME</td>
<td>24</td>
<td>HOME key</td>
</tr>
<tr>
<td>VK_LEFT</td>
<td>25</td>
<td>LEFT ARROW key</td>
</tr>
<tr>
<td>VK_UP</td>
<td>26</td>
<td>UP ARROW key</td>
</tr>
<tr>
<td>VK_RIGHT</td>
<td>27</td>
<td>RIGHT ARROW key</td>
</tr>
<tr>
<td>VK_DOWN</td>
<td>28</td>
<td>DOWN ARROW key</td>
</tr>
<tr>
<td>VK_SELECT</td>
<td>29</td>
<td>SELECT key</td>
</tr>
<tr>
<td></td>
<td>2A</td>
<td>Original equipment manufacturer (OEM) specific</td>
</tr>
<tr>
<td>VK_EXECUTE</td>
<td>2B</td>
<td>EXECUTE key</td>
</tr>
<tr>
<td>VK_SNAPSHOT</td>
<td>2C</td>
<td>PRINT SCREEN key for Windows 3.0 and later</td>
</tr>
<tr>
<td>VK_INSERT</td>
<td>2D</td>
<td>INS key</td>
</tr>
<tr>
<td>VK_DELETE</td>
<td>2E</td>
<td>DEL key</td>
</tr>
<tr>
<td>VK_HELP</td>
<td>2F</td>
<td>HELP key</td>
</tr>
<tr>
<td>Symbolic Constant Name</td>
<td>Hexadecimal Value</td>
<td>Mouse or Keyboard Equivalent</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>VK_0</td>
<td>30</td>
<td>0 key</td>
</tr>
<tr>
<td>VK_1</td>
<td>31</td>
<td>1 key</td>
</tr>
<tr>
<td>VK_2</td>
<td>32</td>
<td>2 key</td>
</tr>
<tr>
<td>VK_3</td>
<td>33</td>
<td>3 key</td>
</tr>
<tr>
<td>VK_4</td>
<td>34</td>
<td>4 key</td>
</tr>
<tr>
<td>VK_5</td>
<td>35</td>
<td>5 key</td>
</tr>
<tr>
<td>VK_6</td>
<td>36</td>
<td>6 key</td>
</tr>
<tr>
<td>VK_7</td>
<td>37</td>
<td>7 key</td>
</tr>
<tr>
<td>VK_8</td>
<td>38</td>
<td>8 key</td>
</tr>
<tr>
<td>VK_9</td>
<td>39</td>
<td>9 key</td>
</tr>
<tr>
<td>VK_A</td>
<td>41</td>
<td>A key</td>
</tr>
<tr>
<td>VK_B</td>
<td>42</td>
<td>B key</td>
</tr>
<tr>
<td>VK_C</td>
<td>43</td>
<td>C key</td>
</tr>
<tr>
<td>VK_D</td>
<td>44</td>
<td>D key</td>
</tr>
<tr>
<td>VK_E</td>
<td>45</td>
<td>E key</td>
</tr>
<tr>
<td>VK_F</td>
<td>46</td>
<td>F key</td>
</tr>
<tr>
<td>VK_G</td>
<td>47</td>
<td>G key</td>
</tr>
<tr>
<td>VK_H</td>
<td>48</td>
<td>H key</td>
</tr>
<tr>
<td>VK_I</td>
<td>49</td>
<td>I key</td>
</tr>
<tr>
<td>VK_J</td>
<td>4A</td>
<td>J key</td>
</tr>
<tr>
<td>VK_K</td>
<td>4B</td>
<td>K key</td>
</tr>
<tr>
<td>VK_L</td>
<td>4C</td>
<td>L key</td>
</tr>
<tr>
<td>VK_M</td>
<td>4D</td>
<td>M key</td>
</tr>
<tr>
<td>VK_N</td>
<td>4E</td>
<td>N key</td>
</tr>
<tr>
<td>VK_O</td>
<td>4F</td>
<td>O key</td>
</tr>
<tr>
<td>Symbolic Constant Name</td>
<td>Hexadecimal Value</td>
<td>Mouse or Keyboard Equivalent</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>VK_P</td>
<td>50</td>
<td>P key</td>
</tr>
<tr>
<td>VK_Q</td>
<td>51</td>
<td>Q key</td>
</tr>
<tr>
<td>VK_R</td>
<td>52</td>
<td>R key</td>
</tr>
<tr>
<td>VK_S</td>
<td>53</td>
<td>S key</td>
</tr>
<tr>
<td>VK_T</td>
<td>54</td>
<td>T key</td>
</tr>
<tr>
<td>VK_U</td>
<td>55</td>
<td>U key</td>
</tr>
<tr>
<td>VK_V</td>
<td>56</td>
<td>V key</td>
</tr>
<tr>
<td>VK_W</td>
<td>57</td>
<td>W key</td>
</tr>
<tr>
<td>VK_X</td>
<td>58</td>
<td>X key</td>
</tr>
<tr>
<td>VK_Y</td>
<td>59</td>
<td>Y key</td>
</tr>
<tr>
<td>VK_Z</td>
<td>5A</td>
<td>Z key</td>
</tr>
<tr>
<td></td>
<td>5B-5F</td>
<td>Undefined</td>
</tr>
<tr>
<td>VK_NUMPAD0</td>
<td>60</td>
<td>Numeric keypad 0 key</td>
</tr>
<tr>
<td>VK_NUMPAD1</td>
<td>61</td>
<td>Numeric keypad 1 key</td>
</tr>
<tr>
<td>VK_NUMPAD2</td>
<td>62</td>
<td>Numeric keypad 2 key</td>
</tr>
<tr>
<td>VK_NUMPAD3</td>
<td>63</td>
<td>Numeric keypad 3 key</td>
</tr>
<tr>
<td>VK_NUMPAD4</td>
<td>64</td>
<td>Numeric keypad 4 key</td>
</tr>
<tr>
<td>VK_NUMPAD5</td>
<td>65</td>
<td>Numeric keypad 5 key</td>
</tr>
<tr>
<td>VK_NUMPAD6</td>
<td>66</td>
<td>Numeric keypad 6 key</td>
</tr>
<tr>
<td>VK_NUMPAD7</td>
<td>67</td>
<td>Numeric keypad 7 key</td>
</tr>
<tr>
<td>VK_NUMPAD8</td>
<td>68</td>
<td>Numeric keypad 8 key</td>
</tr>
<tr>
<td>VK_NUMPAD9</td>
<td>69</td>
<td>Numeric keypad 9 key</td>
</tr>
<tr>
<td>VK_MULTIPLY</td>
<td>6A</td>
<td>Multiply key</td>
</tr>
<tr>
<td>VK_ADD</td>
<td>6B</td>
<td>Add key</td>
</tr>
<tr>
<td>VK_SEPARATOR</td>
<td>6C</td>
<td>Separator key</td>
</tr>
<tr>
<td>VK_SUBTRACT</td>
<td>6D</td>
<td>Subtract key</td>
</tr>
<tr>
<td>Symbolic Constant Name</td>
<td>Hexadecimal Value</td>
<td>Mouse or Keyboard Equivalent</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>VK_DECIMAL</td>
<td>6E</td>
<td>Decimal key</td>
</tr>
<tr>
<td>VK_DIVIDE</td>
<td>6F</td>
<td>Divide key</td>
</tr>
<tr>
<td>VK_F1</td>
<td>70</td>
<td>F1 key</td>
</tr>
<tr>
<td>VK_F2</td>
<td>71</td>
<td>F2 key</td>
</tr>
<tr>
<td>VK_F3</td>
<td>72</td>
<td>F3 key</td>
</tr>
<tr>
<td>VK_F4</td>
<td>73</td>
<td>F4 key</td>
</tr>
<tr>
<td>VK_F5</td>
<td>74</td>
<td>F5 key</td>
</tr>
<tr>
<td>VK_F6</td>
<td>75</td>
<td>F6 key</td>
</tr>
<tr>
<td>VK_F7</td>
<td>76</td>
<td>F7 key</td>
</tr>
<tr>
<td>VK_F8</td>
<td>77</td>
<td>F8 key</td>
</tr>
<tr>
<td>VK_F9</td>
<td>78</td>
<td>F9 key</td>
</tr>
<tr>
<td>VK_F10</td>
<td>79</td>
<td>F10 key</td>
</tr>
<tr>
<td>VK_F11</td>
<td>7A</td>
<td>F11 key</td>
</tr>
<tr>
<td>VK_F12</td>
<td>7B</td>
<td>F12 key</td>
</tr>
<tr>
<td>VK_F13</td>
<td>7C</td>
<td>F13 key</td>
</tr>
<tr>
<td>VK_F14</td>
<td>7D</td>
<td>F14 key</td>
</tr>
<tr>
<td>VK_F15</td>
<td>7E</td>
<td>F15 key</td>
</tr>
<tr>
<td>VK_F16</td>
<td>7F</td>
<td>F16 key</td>
</tr>
<tr>
<td>VK_F17</td>
<td>80H</td>
<td>F17 key</td>
</tr>
<tr>
<td>VK_F18</td>
<td>81H</td>
<td>F18 key</td>
</tr>
<tr>
<td>VK_F19</td>
<td>82H</td>
<td>F19 key</td>
</tr>
<tr>
<td>VK_F20</td>
<td>83H</td>
<td>F20 key</td>
</tr>
<tr>
<td>VK_F21</td>
<td>84H</td>
<td>F21 key</td>
</tr>
<tr>
<td>VK_F22</td>
<td>85H</td>
<td>F22 key</td>
</tr>
<tr>
<td>VK_F23</td>
<td>86H</td>
<td>F23 key</td>
</tr>
<tr>
<td>Symbolic Constant Name</td>
<td>Hexadecimal Value</td>
<td>Mouse or Keyboard Equivalent</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>VK_F24</td>
<td>87H</td>
<td>F24 key</td>
</tr>
<tr>
<td></td>
<td>88-8F</td>
<td>Unassigned</td>
</tr>
<tr>
<td>VK_NUMLOCK</td>
<td>90</td>
<td>NUM LOCK key</td>
</tr>
<tr>
<td>VK_SCROLL</td>
<td>91</td>
<td>SCROLL LOCK key</td>
</tr>
<tr>
<td></td>
<td>92-B9</td>
<td>Unassigned</td>
</tr>
<tr>
<td>BA-C0</td>
<td>OEM specific</td>
<td></td>
</tr>
<tr>
<td>C1-DA</td>
<td>Unassigned</td>
<td></td>
</tr>
<tr>
<td>DB-E4</td>
<td>OEM specific</td>
<td></td>
</tr>
<tr>
<td>E5</td>
<td>Unassigned</td>
<td></td>
</tr>
<tr>
<td>E6</td>
<td>OEM specific</td>
<td></td>
</tr>
<tr>
<td>E7-E8</td>
<td>Unassigned</td>
<td></td>
</tr>
<tr>
<td>E9-F5</td>
<td>OEM specific</td>
<td></td>
</tr>
<tr>
<td>F6-FE</td>
<td>Unassigned</td>
<td></td>
</tr>
</tbody>
</table>
Windows Color and Font Support

Color and font support have been implemented for WAVE Widgets on Windows platforms.

To specify widget colors, use the Background, Foreground, and Basecolor keywords as documented for UNIX systems. The following file,

(Windows)  <wavedir>/lib/std/windows/rgb.txt

lists the names of the available colors, where <wavedir> is the main PV-WAVE directory.

**NOTE**  This file in the Windows platform is a reference only; modifying the file won’t affect the colors used by PV-WAVE.

Predefined Colors for Windows Systems

In addition to the colors listed in the rgb.txt file, you may use any of the following predefined Windows system colors as your widget colors.

<table>
<thead>
<tr>
<th>Windows System Colors</th>
<th>Corresponding Widget Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR_3DDKSHADOW</td>
<td>Dark shadow for 3D display elements</td>
</tr>
<tr>
<td>COLOR_3DFACE,</td>
<td>Face color for 3D display elements</td>
</tr>
<tr>
<td>COLOR_BTNFACE</td>
<td></td>
</tr>
<tr>
<td>Windows System Colors</td>
<td>Corresponding Widget Color</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>COLOR_3DHILIGHT,</td>
<td>Highlight color for 3D display elements (for edges facing the light source)</td>
</tr>
<tr>
<td>COLOR_3DHIGHLIGHT,</td>
<td></td>
</tr>
<tr>
<td>COLOR_BTNHILIGHT,</td>
<td></td>
</tr>
<tr>
<td>COLOR_BTNHIGHLIGHT</td>
<td></td>
</tr>
<tr>
<td>COLOR_3DLIGHT</td>
<td>Light color for 3D display elements (for edges facing the light source)</td>
</tr>
<tr>
<td>COLOR_3DSHADOW,</td>
<td>Shadow color for 3D display elements (for edges facing away from the light source)</td>
</tr>
<tr>
<td>COLOR_BTNSHADOW</td>
<td></td>
</tr>
<tr>
<td>COLOR_ACTIVEBORDER</td>
<td>Active window border</td>
</tr>
<tr>
<td>COLOR_ACTIVECAPTION</td>
<td>Active window caption</td>
</tr>
<tr>
<td>COLOR_APPWORKSPACE</td>
<td>Background color of multiple document interface (MDI) applications</td>
</tr>
<tr>
<td>COLOR_BACKGROUND,</td>
<td>Desktop</td>
</tr>
<tr>
<td>COLOR_DESKTOP</td>
<td></td>
</tr>
<tr>
<td>COLOR_BTNTEXT</td>
<td>Text on push buttons</td>
</tr>
<tr>
<td>COLOR_CAPTIONTEXT</td>
<td>Text in caption, size box, and scroll bar arrow box</td>
</tr>
<tr>
<td>COLOR_GRAYTEXT</td>
<td>Grayed (disabled) text; this color is set to 0 if the current display driver does not support a solid gray color.</td>
</tr>
<tr>
<td>COLOR_HIGHLIGHT</td>
<td>Items selected in a control</td>
</tr>
<tr>
<td>COLOR_HIGHLIGHTTEXT</td>
<td>Text of selected items in a control</td>
</tr>
<tr>
<td>COLOR_INACTIVEBORDER</td>
<td>Inactive window border</td>
</tr>
<tr>
<td>COLOR_INACTIVECAPTION</td>
<td>Inactive window caption</td>
</tr>
<tr>
<td>COLOR_INFOBK</td>
<td>Color of text in an inactive caption</td>
</tr>
<tr>
<td>COLOR_INFOTEXT</td>
<td>Text color for tool tip controls</td>
</tr>
<tr>
<td>COLOR_MENU</td>
<td>Menu background</td>
</tr>
<tr>
<td>COLOR-Menutext</td>
<td>Text in menus</td>
</tr>
<tr>
<td>COLOR_SCROLLBAR</td>
<td>Scroll bar gray area</td>
</tr>
<tr>
<td>COLOR_WINDOW</td>
<td>Window background</td>
</tr>
</tbody>
</table>
Specifying \texttt{XtDefaultForeground} or \texttt{XtDefaultBackground} causes the widget to use the appropriate Windows system colors.

### Setting Fonts on Windows

On Windows, use the \texttt{MSFont} keyword to specify fonts for use in WAVE Widgets. The \texttt{MSFont} keyword is specified as a string of the following form:

\[
\texttt{MSFont} = \text{'face\_name, point\_size, attribute'}
\]

where \texttt{face\_name} specifies the type face, and \texttt{attribute} specifies font attributes such as bold, italic, underline, etc.

For example: \texttt{MSFont} = \text{'Arial, 8, bold'}

You can also specify Windows system fonts as shown in the following table.

<table>
<thead>
<tr>
<th>Windows System Fonts</th>
<th>Corresponding Widget Fonts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI_FIXED</td>
<td>Windows fixed-pitch (monospace) system font.</td>
</tr>
<tr>
<td>ANSI_VAR</td>
<td>Windows variable-pitch (proportional space) system font.</td>
</tr>
<tr>
<td>DEFAULT_GUI</td>
<td>Windows 95/NT 4.0 only: Default font for user interface objects such as menus and dialog boxes.</td>
</tr>
<tr>
<td>DEVICE_DEFAULT</td>
<td>Windows NT only: Device-dependent font.</td>
</tr>
<tr>
<td>OEM_FIXED</td>
<td>Original equipment manufacturer (OEM) dependent fixed-pitch (monospace) font.</td>
</tr>
<tr>
<td>SYSTEM_VAR</td>
<td>By default, Windows uses the system font to draw menus, dialog box controls, and text. In Windows versions 3.0 and later, the system font is a proportionally spaced font; earlier versions of Windows used a monospace system font.</td>
</tr>
</tbody>
</table>
If the font you specify isn’t supported on your system, Windows substitutes another font, usually the system font.

<table>
<thead>
<tr>
<th>Windows System Fonts</th>
<th>Corresponding Widget Fonts</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM_FIXED</td>
<td>Fixed-pitch (monospace) system font used in Windows versions earlier than 3.0. This font is provided for backwards compatibility with earlier versions of Windows.</td>
</tr>
</tbody>
</table>
Index

A
accessing data in PV-WAVE (Windows) 109
annotation
  embed in text grael 496
applications, noninteractive 57, 114
attributes
  resource file 204

B
Bourne shell 3, 9

C
C programming language. See format strings; interapplication communication
C programs
  accessing PV-WAVE’s variable data space from 109–110
  running PV-WAVE from (Windows) 100
C shell 9
callback
  See also Widget Toolbox
  adding to a widget 222
  definition of 164, 222
  for menu button 176
  list 222
  parameters, Motif B-2–B-5
  reason 222
calling PV-WAVE. See interapplication communication
CALL_UNIX function 59–76
CALL_WAVE procedure 68–77
CD procedure 13
child processes, spawning 19–20

See also interapplication communication; servers; sockets
C program as 77
definition of 59
linking with PV-WAVE 61
PV-WAVE as (CALL_UNIX) 61, 73–76, 80
closing
  socket connection 89
  widget hierarchy 207, 210, 221
color
  in resource file 165
  rgb.txt F-1
  setting colors for WAVE Widgets 201
  specifying by name 201
  Windows system F-1
command interpreter, VMS 9
common block
  not needed with user data 207
  used to pass client data 222–225
compiling
  external programs 22
  under UNIX for LINKNLOAD 33, 35
console, Windows 119
cursor
  custom C-4
  used with WtCursor C-1
cwavec routine 37–42, 99
See also interapplication communication
cwavefor routine 37–42, 99
See also interapplication communication
**D**

Data

accessing with wavevars 50, 109

dde

overview 116

server, initializing 116

topics 117

using 116

dialog box

alert popup 233

generic 262

dll file, building 112

dynamic

linking 113

**G**

Grael. See VDA Tools, graphical elements

Graphical element. See VDA Tools, graphical elements

Graphical user interface development. See VDA Tools, WAVE Widgets, Widget Toolbox

GUI development. See graphical user interface development

**H**

Help, online

adding to 155

in VDA Tools 152

Home window

localizing 584

**I**

Input/output

capturing output 12

UNIX pipes 19

Interapplication communication

accessing external functions 27–36

Bidirectional and unidirectional 17

C programs

accessing variable data space of PV-WAVE from 50–51

as a client 77

as a server 75

calling from PV-WAVE 20

calling PV-WAVE from 24, 40

CALL_WAVE 70

CALL_WAVE as a client 77

end session with waveterm 24

linking to PV-WAVE 24, 48, 50

loading a UT_VAR 81, 84

running PV-WAVE from 39, 44, 104

sending commands to PV-WAVE 24

SETBUF function 20–21

used with CALL_UNIX 62

w_cmpnd_reply 63

**F**

Files

See also Data; input/output; Reading; Writing

Fonts

setting fonts for WAVE Widgets 203

specifying by name 203

Windows system F-3

FORTRAN programming language

See also Format strings; interapplication communication
with a child process 19
with C application 118
with RPCs (remote procedure calls) 59

internationalization 581
interprocess communication. See
interapplication communication

K
Korn shell 9

L
linking. See interapplication communication
LINKNLOAD function 18, 27
list widget, managing 287
LoadWave() 113
localization 581
logical names
deleting 6
returning 5
setting 5
longword
integer, on Digital UNIX 28
loop, event. See event loop

M
makefile 48
See also interapplication communication
menus, creating 173–179
message
popup 191
methods, definition of 131
MSFont keyword 204, F-3

N
Navigator
See also VDA Tools
creating 146, 147
directory structure 146
hierarchy in VDA Tool architecture 124
introduction 146
main program file 147
menu bar 146
modifying 147
starting VDA Tools 123
VDA Tool interface 121
VDA Tools 146
X resources 148
nonblocking Widgets 114
noninteractive applications 57, 114

O

object modules 22
See also interapplication communication
operating system
accessing using SPAWN 7
communicating with 1
OPI applications
Windows 113

P

pipes, UNIX 19
See also interapplication communication;
UNIX operating system
POPD procedure 13
print dialog box, localizing 586
PRINTD procedure 13
processes
spawning 7–8
programming
setting resources 205
tips 231
WAVE Widgets 162
Widget Toolbox 217
PUSHD procedure 13

R

resizing windows 115
resource
defining 220
file 165, 219
how to use 204
introduction 148
setting for Widget Toolbox 220
using in PV-WAVE 150
value data type 221
rgb.txt F-1
RPC
See also interapplication communication
description of 59
element, CALL_UNIX 73–80
on Solaris systems 73
synchronization of processes 60

S

separator, widget 332
servers
See also client; interapplication communication
C program 75
definition of 59
element program (test_server.c) 67
linking with PV-WAVE 61
PV-WAVE as 68–70, 84–85
UNIX_REPLY function 72
setbuf() C function 20
signal processing
See also filters; highpass filters;
lowness filters
SPARCworks compiler 47
SPAWN procedure 7, 17
See also interapplication communication
avoiding shell processes 9
when to use 94
spawning a process 7–12
starting PV-WAVE
from an external program 23
statically linked programs 37, 98
stdio facility 20
strings
introduction 148
key names 386
string files 150
using in PV-WAVE 150
STRLOOKUP function 151
subsystem, Windows 119
symbols
VMS
deleting 7
setting 6
<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>TmAddGrael procedure</td>
<td>473</td>
</tr>
<tr>
<td>TmAddSelectedGrael procedure</td>
<td>474</td>
</tr>
<tr>
<td>TmAddSelectedVars procedure</td>
<td>413</td>
</tr>
<tr>
<td>TmAddVar procedure</td>
<td>415</td>
</tr>
<tr>
<td>TmAxis procedure</td>
<td>475</td>
</tr>
<tr>
<td>TmBitmap procedure</td>
<td>477</td>
</tr>
<tr>
<td>TmBottomGrael procedure</td>
<td>478</td>
</tr>
<tr>
<td>TmCodeGen procedure</td>
<td>416</td>
</tr>
<tr>
<td>TmCopy procedure</td>
<td>417</td>
</tr>
<tr>
<td>TmCut procedure</td>
<td>418</td>
</tr>
<tr>
<td>TmDelete procedure</td>
<td>420</td>
</tr>
<tr>
<td>TmDelGrael procedure</td>
<td>479</td>
</tr>
<tr>
<td>TmDelSelectedGrael procedure</td>
<td>480</td>
</tr>
<tr>
<td>TmDelVar procedure</td>
<td>419</td>
</tr>
<tr>
<td>TmDeselectVars procedure</td>
<td>421</td>
</tr>
<tr>
<td>TmDynamicDisplay procedure</td>
<td>423</td>
</tr>
<tr>
<td>TmEnumerateAttributes procedure</td>
<td>426</td>
</tr>
<tr>
<td>TmEnumerateGrael function</td>
<td>482</td>
</tr>
<tr>
<td>TmEnumerateGraelMethods function</td>
<td>481</td>
</tr>
<tr>
<td>TmEnumerateItems procedure</td>
<td>428</td>
</tr>
<tr>
<td>TmEnumerateMethods function</td>
<td>429</td>
</tr>
<tr>
<td>TmEnumerateSelectedGraels function</td>
<td>483</td>
</tr>
<tr>
<td>TmEnumerateSelectedVars function</td>
<td>430</td>
</tr>
<tr>
<td>TmEnumerateToolNames function</td>
<td>431</td>
</tr>
<tr>
<td>TmEnumerateVars function</td>
<td>432</td>
</tr>
<tr>
<td>TmExecuteGraelMethod procedure</td>
<td>484</td>
</tr>
<tr>
<td>TmExecuteMethod procedure</td>
<td>433</td>
</tr>
<tr>
<td>TmExport procedure</td>
<td>435</td>
</tr>
<tr>
<td>TmExportSelection procedure</td>
<td>436</td>
</tr>
<tr>
<td>TmGetAttribute function</td>
<td>437</td>
</tr>
<tr>
<td>TmGetGraelMethod function</td>
<td>485</td>
</tr>
<tr>
<td>TmGetGraelRectangle function</td>
<td>486</td>
</tr>
<tr>
<td>TmGetMessage function</td>
<td>151, 439</td>
</tr>
<tr>
<td>TmGetMethod function</td>
<td>441</td>
</tr>
<tr>
<td>TmGetTop function</td>
<td>442</td>
</tr>
<tr>
<td>TmGetUniqueGraelName function</td>
<td>487</td>
</tr>
<tr>
<td>TmGetUniqueToolName function</td>
<td>126, 443</td>
</tr>
<tr>
<td>TmGetVarMainName function</td>
<td>444</td>
</tr>
<tr>
<td>TmGroupGraels function</td>
<td>488</td>
</tr>
<tr>
<td>TmInit procedure</td>
<td>445</td>
</tr>
<tr>
<td>TmLegend procedure</td>
<td>489</td>
</tr>
<tr>
<td>TmLine procedure</td>
<td>490</td>
</tr>
<tr>
<td>TmList function</td>
<td>446</td>
</tr>
<tr>
<td>TmListAppend procedure</td>
<td>447</td>
</tr>
<tr>
<td>TmListClear procedure</td>
<td>448</td>
</tr>
<tr>
<td>TmListDelete procedure</td>
<td>449</td>
</tr>
<tr>
<td>TmListDestroy procedure</td>
<td>450</td>
</tr>
<tr>
<td>TmListExtend procedure</td>
<td>451</td>
</tr>
<tr>
<td>TmListGetMethod function</td>
<td>452</td>
</tr>
<tr>
<td>TmListInsert procedure</td>
<td>455</td>
</tr>
<tr>
<td>TmListReplace procedure</td>
<td>456</td>
</tr>
<tr>
<td>TmListRetrieve function</td>
<td>458</td>
</tr>
<tr>
<td>TmListSetMethod procedure</td>
<td>459</td>
</tr>
<tr>
<td>TmPaste procedure</td>
<td>461</td>
</tr>
<tr>
<td>TmRect procedure</td>
<td>492</td>
</tr>
<tr>
<td>TmRegister procedure</td>
<td>462</td>
</tr>
<tr>
<td>TmRestoreTemplate function</td>
<td>463</td>
</tr>
<tr>
<td>TmRestoreTools function</td>
<td>464</td>
</tr>
<tr>
<td>TmSaveTools procedure</td>
<td>465</td>
</tr>
<tr>
<td>TmSetAttribute function</td>
<td>466</td>
</tr>
<tr>
<td>TmSetGraelMethod procedure</td>
<td>493</td>
</tr>
<tr>
<td>TmSetGraelRectangle procedure</td>
<td>494</td>
</tr>
<tr>
<td>TmSetMethod procedure</td>
<td>468</td>
</tr>
<tr>
<td>TmStartCodegen procedure</td>
<td>469</td>
</tr>
<tr>
<td>TmText procedure</td>
<td>495</td>
</tr>
<tr>
<td>TmTopGrael procedure</td>
<td>497</td>
</tr>
<tr>
<td>TmUngroupGraels procedure</td>
<td>498</td>
</tr>
<tr>
<td>TmUnregister procedure</td>
<td>470</td>
</tr>
<tr>
<td>TM_CODEGEN method</td>
<td>133</td>
</tr>
<tr>
<td>TM_CONVERT method</td>
<td>133</td>
</tr>
<tr>
<td>TM_DATA_EXPORT method</td>
<td>133</td>
</tr>
<tr>
<td>TM_DATA_IMPORT method</td>
<td>133</td>
</tr>
<tr>
<td>TM_DATA_SELECTION method</td>
<td>133</td>
</tr>
<tr>
<td>TM_DISPLAY method</td>
<td>133</td>
</tr>
<tr>
<td>TM_VARATTRIBUTES method</td>
<td>133</td>
</tr>
<tr>
<td>TM_VIEWATTRIBUTES method</td>
<td>133</td>
</tr>
</tbody>
</table>

**U**

UNIX operating system
- avoiding shells with SPAWN procedure 11
- CALL_UNIX 59
- commands from within PV-WAVE 3–7
- compiling for LINKNLOAD 33, 35
- environment, description of 2
- pipe, description of 19
- shells 9
- UNIX_LISTEN function 68, 71, 84
UNIX_REPLY function 68, 72, 84
unnamed structures
defining menu items with 176–177
uses for 220
UT_VAR 81, 84

V

variables
passing 207
VDA Tools
See also VDA Tools Manager; VDA Tools Utilities; Navigator
advantages 160
architecture 124
attributes, getting and setting 134
components of 130
directory structure 126
disadvantages 160
exporting variables from 141
graphical elements
API routines 138
definition of 136
embedded text in text grael 496
manipulating 137
intertool communication 123, 141
items
definition of 134
list of standard 135
localizing 581
main features 121–124
methods
definition of executing 131
definition of registering 131
list of standard 132
TM_CODEGEN 133
TM_CONVERT 133
TM_DATA_EXPORT 133
TM_DATA_IMPORT 133
TM_DATA_SELECTION 133
TM_DISPLAY 133
TM_VAR_ATTRIBUTES 133
TM_VIEW_ATTRIBUTES 133
online help 124, 140
unique names of 130
user interface 139
variables, manipulating 134
VDA Tools Manager
See also VDA Tools Utilities; VDA Tools

adding graphical elements
to graphical element list 473
to graphical element selection list 474
adding variables
to a VDA Tool 415
to selected variables list 473
API, functional listing 129
attributes
enumerating 426
getting value for 437
listing 426
setting 466
code generation file
closing 425
opening 469
writing 416
coordinates, of rectangular boundary 486
copying graphics to clipboard 417
deleting
graphical element from associated list 479
graphical element from selected list 480
graphics 418, 420
variables 419, 421
VDA Tool from registry 470
display list
bottom graphical element 478
top graphical element 497
executing
a method 433, 484
graphical element method 484
exporting
variable selection list 436
variables 435
files
closing code generation 425
code generation 416
opening for code generation 469
functions and procedures, core 413
graphical elements 473
grouping graphical elements 488
initializing 445
items
enumerating 428
listing 428
See also VDA Tools Manager; VDA Tools

adding
- buttons to button bar 499
- color table button 513
- dialog box 538
- message text 502

button bar
- adding buttons 499
- button sensitivity 511
- changing a button’s setting 510
- creating 506

color index
- getting from a color button 517
- setting 518

color table button 513

definition of 139

dialog box
- creating 538
- restoring status 530
- saving status 530

file checking 512

font option menu 533–537

linestyle option menu 547, 550

loading resources and strings 551, 554

location of
- menu bar 556
- fonts 533
- item sensitivity 563
- linestyles 547, 550
- variables 575, 578

messages
- adding hardcoded text 502
- creating area for 566
- dialog box status 530
- stored in resource files 551

resource files, loading for VDA tools 551, 554

setting
- button sensitivity 511
- color index 518
- fonts in option menu 537
- sensitivity of a menu item 563

text widgets 544

variables
- creating menu of 575

VDA Tools Utilities
getting from menu 577
setting option menu 578
virtual keys (Windows) E-1
VMS operating system
  command interpreter 9
  commands from within PV-WAVE 5–7
  logicals and symbols 5–6

W

WAVE Widgets
See also callback; Widget Toolbox
active buttons for callback 176
alert box
  creating 233
  destroying 237
application example 211
arranging layout 169
attaching widgets 170–172
blocking vs. nonblocking window 194
bulletin board layout 170
button box 179, 238
buttons, event handling 267
color support for F-1 combining with Widget Toolbox 218
command widget 243
  creating basic steps for 163
  command widget 198
drawing area 184, 255
graphical icon buttons 176, 179, 358
  hiding 209
  hierarchy
definition of 166
displaying 210
ID representation 167
icon use in menu 178
ID representation 167
initializing 164, 276
introduction to 161
keys, ASCII values 268
layout 166, 169–173, 279
list of widget types 167–168
loading resources 151
locating selected point coordinate 270
location in directory 161
managing widgets
  destroying the shell 207
  show/hide 209
  user data 207
message buttons 192
Motif GUI 157
multi-line text window 188
option menu 175
passing user-defined data 207
popup menu 174, 176, 312
popup widget 169, 191, 194, 196, 199
preview widget 315, 324
radio box buttons 181
resource database for widgets 330
scrolling list 189–190, 283
sensitivity 209
separator 332
setting
  colors 201
  fonts 203
  resources 205
  values 206
showing 209
single-line editable field 187
single-line read-only label 187
sliders 182–183, 247
submitting routines to Users’ Library 162
table widget 200
text widget 352
timer registration 357
toggle buttons in menus 176
toolbox buttons 179
user data
defining and passing 207
destroying the shell 207
widget definition 161
Windows GUI 157
with OpenVMS 24-bit display hardware 157
wavecmd routine 24–26
waveinit routine 23
wavevars function
  accessing data in PV-WAVE 50, 109
description of 97
examples 53–57, 112
parameters 51, 109
syntax 29, 97, 109
Widget Toolbox
  See also callback; WAVE Widgets
adding
  callbacks 222
event handlers 223, 366
timers 225
callback parameters for Motif B-1
class 219
combining with WAVE Widgets 218
convenience widgets, Motif A-3
creating widgets 219
cursors C-1–C-5
defining keyboard keys 386
event handler 223, 366
event loop 57, 114, 228, 394
example application 229
hierarchy
  destroying and closing 221
displaying 221
ID representation 219
include files 229
initializing 219
loading resources 151
managing widgets 221
pointer function 224, 390
preview window 392
realizing a widget 221
resource database for widgets 396
resources
  getting 220
  retrieving 374
  setting 220, 398
  running an application 228
24-bit hardware 218
widget classes, Motif A-1

Windows events 366
X events 366
Widget. See WAVE Widgets; Widget Toolbox
window
  index 184
  resizing 59, 115
Windows F-1
c потом нормально
  GUI development 157
  interapplication communication 93
  system color F-1
  system fonts F-3
WoAddButtons procedure 499
WoAddMessage procedure 502
WoAddStatus function 503
WoBuildResourceFilename function 504
WoButtonBar function 506
WoButtonBarSet procedure 510
WoButtonBarSetSensitivity procedure 511
WoCheckFile function 512
WoColorButton function 513
WoColorButtonGetValue function 517
WoColorButtonSetValue procedure 518
WoColorConvert function 518
WoColorGrid procedure 520
WoColorGridGetValue procedure 523
WoColorGridSetValue procedure 524
WoColorWheel function 525
WoConfirmClose procedure 528
WoDialogStatus procedure 530
WoFontOptionMenu function 533
WoFontOptionMenuGetValue function 536
WoFontOptionMenuSetValue function 537
WoGenericDialog function 538
WoGetToolNameFromTitle function 542
WoGetUniqueWindowTitle function 543
WoLabeledText function 544
WoLineStyleOptionMenu function 547
WoLineStyleOptionMenuGetValue function 549
WoLineStyleOptionMenuSetValue procedure 550
WoLoadResources procedure 151, 551
WoLoadStrings procedure 151, 554
WoMenuBar function 556
WoMenuBarSetSensitivity function 563
WoMenuBarSetToggle procedure 565
WoMessage function 566
WoSaveAsPixmap procedure 568
WoSetCursor procedure 569
WoSetToolIcon procedure 570
WoSetWindowTitle procedure 572
WoStatus function 573
WoVariableOptionMenu function 575
WoVariableOptionMenuGetValue function 577
WoVariableOptionMenuSetValue procedure 578
WtAddCallback function 222, 363
WtAddHandler function 223, 365
WtClose function 221, 368
WtCreate function 218, 219, 370
WtCursor function 374
WtList function 383
WtLookup function 383
WtLookupString function 386
WtLoop function 228
WtLoop procedure 387
WtMainLoop function 389
WtPointer function 390
WtPreview function 392
WtProcessEvent function 394
WtResource function 151, 396
WtSet function 218, 398
WtTable function 401
WtTimer function 225, 408
WtWorkProc function 410
WwAlert function 233
WwAlertPopdown procedure 237
WwButtonBox function 180, 238
WwCallback function 242
WwCommand function 198, 243
WwControlsBox function 183, 247
WwDialog function 194, 252
WwDrawing function 184, 255
WwFileSelection function 196, 259
WwGenericDialog function 262
WwGetButton function 267
WwGetKey function 268
WwGetPosition function 270
WwGetValue function 206, 272
WwHandler function 275
WwInit function 164, 276
WwLayout function 169, 279
WwList function 189, 283
WwListUtils function 287
WwLoop function 164, 290
WwMainWindow function 166, 291
WwMenuBar function 174, 294
WwMenuItem function 298
WwMenuBar function 174, 294
WwMultiClickHandler function 304
WwOptionMenu function 175, 306
WwPickFile function 310
WwPopupMenu function 175, 312
WwPreview function 315
WwPreviewUtils function 324
WwRadioBox function 181, 325
WwResource function 330
WwSeparator function 332
WwSetCursor function 335
WwSetValue function 206, 336
WwTable function 339
WwTableUtils function 346
WwText function 186, 352
WwTimer function 357
WwToolBox function 180, 358
w_cmpnd_reply function 63, 65
w_get_par function 64
w_send_reply function 64
w_smpl_reply function 63, 65

X

X server connecting to 164
X Window System resources 204
.Xdefaults file 205
Xlib 158
Xt Intrinsics 158, 164, 219
Xt TimeOut function 224