



User Interface Programming

IDL Version 7.0
November 2007 Edition
Copyright © ITT Visual Information Solutions
All Rights Reserved

Restricted Rights Notice

The IDL®, IDL Analyst™, ENVI®, and ENVI Zoom™ software programs and the accompanying procedures, functions, and documentation described herein are sold under license agreement. Their use, duplication, and disclosure are subject to the restrictions stated in the license agreement. ITT Visual Information Solutions reserves the right to make changes to this document at any time and without notice.

Limitation of Warranty

ITT Visual Information Solutions makes no warranties, either express or implied, as to any matter not expressly set forth in the license agreement, including without limitation the condition of the software, merchantability, or fitness for any particular purpose.

ITT Visual Information Solutions shall not be liable for any direct, consequential, or other damages suffered by the Licensee or any others resulting from use of the software packages or their documentation.

Permission to Reproduce this Manual

If you are a licensed user of these products, ITT Visual Information Solutions grants you a limited, nontransferable license to reproduce this particular document provided such copies are for your use only and are not sold or distributed to third parties. All such copies must contain the title page and this notice page in their entirety.

Export Control Information

This software and its associated documentation are subject to the controls of the Export Administration Regulations (EAR). It has been determined that this software is classified as EAR99 under U.S. Export Control laws and regulations, and may not be re-transferred to any destination expressly prohibited by U.S. laws and regulations. The recipient is responsible for ensuring compliance to all applicable U.S. Export Control laws and regulations.

Acknowledgments

ENVI® and IDL® are registered trademarks of ITT Corporation, registered in the United States Patent and Trademark Office. ION™, ION Script™, ION Java™, and ENVI Zoom™ are trademarks of ITT Visual Information Solutions.

Numerical Recipes™ is a trademark of Numerical Recipes Software. Numerical Recipes routines are used by permission.

GRG2™ is a trademark of Windward Technologies, Inc. The GRG2 software for nonlinear optimization is used by permission.

NCSA Hierarchical Data Format (HDF) Software Library and Utilities. Copyright © 1988-2001, The Board of Trustees of the University of Illinois. All rights reserved.

NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities. Copyright © 1998-2002, by the Board of Trustees of the University of Illinois. All rights reserved.

CDF Library. Copyright © 2002, National Space Science Data Center, NASA/Goddard Space Flight Center.

NetCDF Library. Copyright © 1993-1999, University Corporation for Atmospheric Research/Unidata.

HDF EOS Library. Copyright © 1996, Hughes and Applied Research Corporation.

SMACC. Copyright © 2000-2004, Spectral Sciences, Inc. and ITT Visual Information Solutions. All rights reserved.

This software is based in part on the work of the Independent JPEG Group.

Portions of this software are copyrighted by DataDirect Technologies, © 1991-2003.

BandMax®. Copyright © 2003, The Galileo Group Inc.

Portions of this computer program are copyright © 1995-1999, LizardTech, Inc. All rights reserved. MrSID is protected by U.S. Patent No. 5,710,835. Foreign Patents Pending.

Portions of this software were developed using Unisearch's Kakadu software, for which ITT has a commercial license. Kakadu Software. Copyright © 2001. The University of New South Wales, UNSW, Sydney NSW 2052, Australia, and Unisearch Ltd, Australia.

This product includes software developed by the Apache Software Foundation (www.apache.org/).

MODTRAN is licensed from the United States of America under U.S. Patent No. 5,315,513 and U.S. Patent No. 5,884,226.

FLAASH is licensed from Spectral Sciences, Inc. under a U.S. Patent Pending.

Portions of this software are copyrighted by Merge Technologies Incorporated.

Support Vector Machine (SVM) is based on the LIBSVM library written by Chih-Chung Chang and Chih-Jen Lin (www.csie.ntu.edu.tw/~cjlin/libsvm/), adapted by ITT Visual Information Solutions for remote sensing image supervised classification purposes.

IDL Wavelet Toolkit Copyright © 2002, Christopher Torrence.

IMSL is a trademark of Visual Numerics, Inc. Copyright © 1970-2006 by Visual Numerics, Inc. All Rights Reserved.

Other trademarks and registered trademarks are the property of the respective trademark holders.



Contents

Chapter 1	
Overview of User Interface Options	7
User Interface Options in IDL	8
Creating an iTool Interface	9
Creating a Widget Interface	10
Creating a Custom iTool Interface	11
Chapter 2	
Creating Widget Applications	13
About Widgets	14
About Widget Applications	15
Types of Widgets	16
Widget Programming Concepts	18
Example: A Simple Widget Application	21
Widget Application Lifecycle	23
Manipulating Widgets	26

Working With Widget IDs	31
Widget User Values	33
Widget Event Processing	34
Example: Event Processing and User Values	40
Managing Application State	42
Creating a Compound Widget	46
Example: Compound Widget	49
Debugging Widget Applications	53
Chapter 3	
Widget Application Techniques	55
Working with Widget Events	56
Using Multiple Widget Hierarchies	61
Creating Menus	64
Widget Sizing	76
Tips on Creating Widget Applications	82
Enhancing Widget Application Usability	84
Chapter 4	
Using Widget Buttons	101
Using Button Widgets	102
Bitmap Button Labels	103
Tooltips	106
Exclusive and Non-Exclusive Buttons	107
Chapter 5	
Using Draw Widgets	109
Using Draw Widgets	110
Using Direct Graphics in Draw Widgets	111
Using Object Graphics in Draw Widgets	112
Scrolling Draw Widgets	113
Context Events in Draw Widgets	117
Draw Widget Example	118
Accessing Draw Widget Events	119
Implementing Drag and Drop Functionality	121

Chapter 6	
Using Property Sheet Widgets	125
Using Property Sheet Widgets	126
Registering Properties	127
Selecting Properties	128
Changing Properties	131
User-defined Properties	133
Property Sheet Sizing	134
Property Sheet Example	136
Multiple Properties Example	150
Chapter 7	
Using Table Widgets	153
Using Table Widgets	154
Default Table Size	155
Selection Modes	156
Data Types	158
Data Retrieval	159
Edit Mode	162
Cell Attributes	163
Example: Single Data Type Data	170
Example: Structure Data	174
Chapter 8	
Using Tab Widgets	177
Using Tab Widgets	178
Example: A Simple Tab Widget	179
Tab Sizing and Multiline Behavior	180
Example: Retrieving Values	182
Chapter 9	
Using Tree Widgets	185
Using Tree Widgets	186
Types of Tree Widgets	187
Example: A Simple Tree	188
Setting the Tree Selection State	189
Making a Tree Entry Visible	190

Replacing the Default Bitmaps	191
Dragging and Dropping Tree Nodes	193
Tree Widget Drag and Drop Examples	205
Positioning Tree Nodes	207
Index	209



Chapter 1

Overview of User Interface Options

The following topics are covered in this chapter:

User Interface Options in IDL	8	Creating a Widget Interface	10
Creating an iTool Interface	9	Creating a Custom iTool Interface	11

User Interface Options in IDL

When creating a user-interface in IDL, you have several choices. In order of increasing complexity, you can use any of the following:

- IDL command-line interface — using the IDL command line as a non-graphical user interface to display Direct graphics visualizations, or data in the IDL output log.
- Existing iTool Interface — using an existing iTool provides quick data display and manipulation capabilities for image, plot, surface, volume and map data. See [“Creating an iTool Interface”](#) on page 9 for more information.
- Custom Widget Interface — using widgets offers complete control over user interface design. However, in a traditional widget application, you must code all underlying functionality. There is an option of creating a hybrid widget-iTool application, but this requires additional programming expertise. See [“Creating a Widget Interface”](#) on page 10 for more information.
- Custom iTool — using a custom iTool interface allows you to expand on the capabilities of the standard iTool design, and configure the appearance of the tool. This requires the most programming expertise of the three options. It is likely that one of the other two options will meet the needs of the majority of the applications, but this level of customization is available for those who require it. See [“Creating a Custom iTool Interface”](#) on page 11 for more information.

Creating an iTool Interface

Using an existing iTool user interface for data display and modification is the easiest way to allow your user to access, visualize and modify supported plot, volume, surface, map and image data. See [Chapter 1, “Introducing the IDL iTools”](#) (*iTool User’s Guide*) for information on using the iTools.

If you need functionality beyond that provided by an existing iTool, you can expand the functionality by adding:

- Custom operations or manipulators to standard visualization types
- Custom file writers or file readers
- Custom messages

Using an existing iTool lets to provide your users with a great deal of pre-built functionality. For information on expanding the iTool functionality mentioned above, see the following sections in the *iTool Programming*:

- [Chapter 7, “Creating an Operation”](#) and [Chapter 8, “Creating a Manipulator”](#)
- [Chapter 9, “Creating a File Reader”](#) and [Chapter 10, “Creating a File Writer”](#)
- [Chapter 12, “Using iTool User Interface Elements”](#)

Creating a Widget Interface

IDL allows you to construct and manipulate graphical user interfaces using *widgets*. Widgets (or *controls*, in the terminology of some development environments) are simple graphical objects such as pushbuttons or sliders that allow user interaction via a pointing device (usually a mouse) and a keyboard. Consider developing a widget application when you want complete control over the interface layout and the available UI elements, or when you want to design a workflow of data modification. In addition to this chapter ([Chapter 1, “Overview of User Interface Options”](#)), see the following for more information on creating a widget application:

- [Chapter 3, “Widget Application Techniques”](#)
- [Chapter 4, “Using Widget Buttons”](#)
- [Chapter 5, “Using Draw Widgets”](#)
- [Chapter 6, “Using Property Sheet Widgets”](#)
- [Chapter 7, “Using Table Widgets”](#)
- [Chapter 8, “Using Tab Widgets”](#)
- [Chapter 9, “Using Tree Widgets”](#)

A widget application can include iTool elements, described in the following section.

Creating a Custom iTool Interface

Each of the standard iTools (such as the iPlot or iImage tools) have the same basic interface style. Beyond adding operations or manipulators, you can further modify the existing iTool interface by adding:

- Modal dialogs, implemented through a user interface service
- iTool panels, which provide a set of controls that are attached to a visualization window and are always available

Beyond this, you also have the option of modifying the standard iTool interface. Standard iTools are constructed of a number of compound widgets designed to work explicitly within the iTool architecture. You can modify the standard iTool interface by creating a custom iTool-widget interface, a hybrid tool that combines traditional widget functionality and iTool compound widgets. This requires knowledge of widget programming, how to create an iTool, how to create a UI service, and how to use the iTool compound widgets. For more information on the previous topics, see the following sections in the *iTool Programming*:

- [Chapter 13, “Creating a User Interface Service”](#)
- [Chapter 14, “Creating a User Interface Panel”](#)
- [Chapter 15, “Creating a Custom iTool Widget Interface”](#)



Chapter 2

Creating Widget Applications

The following topics are covered in this chapter:

About Widgets	14	Widget User Values	33
About Widget Applications	15	Widget Event Processing	34
Types of Widgets	16	Example: Event Processing and User Values	40
Widget Programming Concepts	18	Managing Application State	42
Example: A Simple Widget Application ...	21	Creating a Compound Widget	46
Widget Application Lifecycle	23	Example: Compound Widget	49
Manipulating Widgets	26	Debugging Widget Applications	53
Working With Widget IDs	31		

About Widgets

IDL allows you to construct and manipulate graphical user interfaces using *widgets*. Widgets (or *controls*, in the terminology of some development environments) are simple graphical objects such as pushbuttons or sliders that allow user interaction via a pointing device (usually a mouse) and a keyboard. This style of graphical user interaction offers many significant advantages over traditional command-line based systems. (See [Chapter 1, “Overview of User Interface Options”](#) for information on the different types of user interfaces you can create in IDL.)

IDL widgets are significantly easier to use than other alternatives, such as writing a C language program using the native window system graphical interface toolkit directly. IDL handles much of the low-level work involved in using such toolkits. The interpretive nature of IDL makes it easy to prototype potential user interfaces. In addition to the user interface, the author of a program written in a traditional compiled language also must implement any computational and graphical code required by the program. IDL widget programs can draw on the full computational and graphical abilities of IDL to supply these components.

The style of widgets IDL creates depends on the windowing system supported by your host computer. Unix hosts use Motif widgets, while Microsoft Windows systems use the native Windows toolkit. Although the different toolkits produce applications with a slightly different look and feel, most properly-written widget applications work on all systems without change.

IDL graphical user interfaces are constructed by combining widgets in a treelike hierarchy. Each widget has one parent widget and zero or more child widgets. There is one exception: the topmost widget in the hierarchy (called a *top-level base*) is always a base widget and has no parent.

About Widget Applications

The flow of control in a widget application is fundamentally different than in other IDL programs. A program written to be used from the IDL command line generally accepts its inputs when the program is invoked. The program then proceeds in a well-defined order to process those inputs and provide some output — a calculated value, a plot, an image, *etc.* In contrast, widget applications are *event driven*.

In an event driven system, the program creates an interface and then waits for messages (events) to be sent to it from the window system. Events are generated in response to user manipulation, such as pressing a button or moving a slider. The program responds to events by carrying out the action or computation specified by the programmer, and then waiting for the next event.

This approach to computing is fundamentally different from the traditional command-based approach. Actions occur in the order specified by the user at runtime, rather than in the order determined by the programmer. The widget application model and programming techniques are discussed later in this chapter. Events from IDL widgets are generated in the form of an IDL structure variable specific to the widget. Widget events and event-processing are also discussed in detail.

This chapter discusses topics related to creating widget user interfaces, controlling widgets, processing events generated by user interaction, and managing the application state of a widget application. [Chapter 3, “Widget Application Techniques”](#) explores the use of specific types of widgets in widget applications and discusses methods for creating specific types of interfaces and applications.

Running the Example Code

The example code used in this chapter and in [Chapter 3, “Widget Application Techniques”](#) is part of the IDL distribution. All of the examples developed in the text of these chapters are included as `.pro` files in the `examples/doc/widgets` subdirectory of the IDL distribution. By default, this directory is part of IDL’s path; if you have not changed your path, you will be able to run the examples as described here. See [“!PATH” \(IDL Reference Guide\)](#) for information on IDL’s path.

In addition to the examples developed here, a number of simple examples of widget programming can be seen by running the IDL program `wexmaster.pro`, located in the `/examples/widgets/wexmast` folder of the IDL distribution. A widget interface with a pulldown menu of small widget applications should appear.

Types of Widgets

IDL supports several types of widgets and widget-like interface elements that can be used in your widget application:

Type	Descriptions
Widget Primitives	<p><i>Widget primitives</i> are the base interface elements used to create widget applications. They are used to display visualizations, to allow the user to make selections within a UI, and to generate events. IDL widget primitives include standard interface elements such as buttons, combo boxes, lists, tables and labels. You can also add tables, trees, ActiveX controls, drawing areas and property sheets to a widget application.</p> <p>See “Widget Routines” (<i>IDL Quick Reference</i>) for a list of widget primitives and related widget routines.</p>
Compound Widgets	<p><i>Compound widgets</i> are more complex interface elements built from the widget primitives. A compound widget is a complete, self-contained, reusable widget sub-tree that behaves to a large degree just like a widget primitive, but which is written in the IDL language. Compound widgets allow the development of reusable widget code, much like a GUI subroutine.</p> <p>Compound widget routines provided with IDL can be found (along with many other routines that use the widgets) in the <code>lib</code> subdirectory of the IDL distribution. All compound widgets supplied along with IDL have filenames beginning with “<code>CW_</code>” to make them easier to identify.</p> <p>See “Widget Routines, Compound” (<i>IDL Quick Reference</i>) for a list of the compound widget routines provided in IDL.</p> <p>Note - See “Creating a Compound Widget” on page 46 for information on writing your own compound widgets.</p>

Table 2-1: Introduction to Widget Types in IDL

Type	Descriptions
Dialogs	<p><i>Dialogs</i> are widget-like elements that can be called from any IDL application (whether or not it uses other widgets), but which do not belong to a widget hierarchy. Dialogs are useful for informing users of changes in the application state or collecting relatively simple input, such as the answer to a “Yes or No” question or the name of a file. They have short lifetimes, and disappear after serving their purpose.</p> <p>Dialogs are <i>modal</i> (or “blocking”), which means that when a dialog is displayed, no other interface elements (widgets or compound widgets) can be manipulated until the user dismisses the dialog. While the dialog is not part of any widget hierarchy, you can specify a widget over which the dialog will be centered on screen, making it possible to visually associate the dialog with a specific widget application.</p> <p>See “Dialog Routines” (<i>IDL Quick Reference</i>) for a list of available routines.</p>
Utilities	<p><i>Utilities</i> are self-contained widget applications written in the IDL language that can be invoked from the IDL command line or called from within an application. Most names of utility routines are prefaced with the letter “X”.</p> <p>Although utility routines cannot be inserted directly into a widget application (becoming part of the application’s widget hierarchy), they can be linked to a widget application in such a way (via the GROUP keyword) that when the widget application is iconized or destroyed, the utility is iconized or destroyed as well. Utility routines can also be configured as <i>modal</i> applications, requiring that the user exit from the utility before returning to the widget application that called it. See “Using Multiple Widget Hierarchies” in Chapter 3 for further discussion of grouping and modal behaviors.</p> <p>See “Utilities” (<i>IDL Quick Reference</i>) for a list of available routines.</p>

Table 2-1: Introduction to Widget Types in IDL (Continued)

Widget Programming Concepts

This section discusses some basic ideas and concepts that are central to the process of writing IDL widget applications.

Widget Values

Many widget primitives and compound widgets have *widget values* associated with them. Depending on the type of widget, the widget value may represent a static item set by the programmer (the label of a button widget, for example) or a dynamic value set by the user (the numerical value of a slider widget, for example).

Widget values are retrieved from a widget using the `GET_VALUE` keyword to the `WIDGET_CONTROL` procedure, and set either when the widget is created or using the `SET_VALUE` keyword to `WIDGET_CONTROL`. Descriptions of widget value data types and default values are included along with the descriptions of individual widgets in the following sections. (See [“Manipulating Widgets”](#) on page 26 for details on using `WIDGET_CONTROL`.)

Widgets can also have *user values*. A widget’s user value is an IDL variable, and can thus be of any of IDL’s data types. User values can contain any information the programmer wants to include; they are not examined or used by IDL except as specified by the widget application programmer. User values and their role in widget programming are discussed in [“Widget User Values”](#) on page 33.

Note

If a widget value is a string (as for a button label), you can use language catalogs to internationalize the widget with sets of strings in particular languages. For more information, see [“Using Language Catalogs”](#) on page 471.

Widget IDs

IDL widgets are uniquely identified via their *widget IDs*. The widget ID is a long integer assigned to the widget when it is first created; this integer is returned as the value of the widget creation function. For example, you might create a base widget with the following IDL command:

```
base = WIDGET_BASE()
```

Here, the IDL variable `base` receives the widget ID of the newly-created top-level base widget.

Routines within your widget application that need to retrieve data from widgets or change their appearance need access to the widgets' IDs. Techniques for passing widget IDs between independent routines in your widget application are discussed in [“Working With Widget IDs”](#) on page 31.

Widget Parent/Child Relationships

With one exception (described below), when you create a new widget using one of the `WIDGET_*` functions, you specify the widget ID of the new widget's *parent widget*. This parent-child relationship defines a *widget hierarchy*.

For example, suppose you have created a base widget whose widget ID is contained in the IDL variable `base`. The following IDL command creates a button widget that is a *child* of the base widget whose widget ID is stored in the variable `base`:

```
button1 = WIDGET_BUTTON(base, VALUE='Test button')
```

In addition to being below `base` in the widget hierarchy, `button1` appears inside `base1` when the base widget is realized on the screen.

The exception to this parent-child rule is a special instance of a base widget called a *top-level base*. A top-level base is different from an “ordinary” base widget in the following ways:

- It does not have a parent widget
- It serves as the top of a widget hierarchy
- Its widget ID is included in the `TOP` field of every widget event structure generated by other widgets in its hierarchy

In practice, a widget application always begins with a top-level base. The fact that the widget ID of the top-level base widget is always available in the event structure of widget events is very useful for managing the state of a widget application. This topic is discussed in depth in [“Managing Application State”](#) on page 42.

Instantiating and Displaying Widgets

When you call a routine that creates a widget, IDL “creates” the widget and assigns it a unique identifier (the *widget ID*). For example, the following IDL statements create a base widget that holds a button widget, and stores the widgets' identifiers in the variables `base` and `button`:

```
base = WIDGET_BASE()  
button = WIDGET_BUTTON(base, VALUE='My Button')
```

At this point, the widgets are nothing more than data structures (referred to as *widget records*) in IDL's memory. Nothing appears on screen, and in fact IDL has yet to calculate the sizes of the widgets or the way they will appear.

In order to instantiate the widget — that is, to create the final form of the widget that will be displayed from components supplied by the platform-specific user interface toolkit and (in most cases) make it appear on screen — the widgets must be *realized*. Realization occurs with a call to the `WIDGET_CONTROL` procedure, using the `REALIZE` keyword:

```
WIDGET_CONTROL, base, /REALIZE
```

After this command has been issued, the widgets appear on the computer screen. (See [“Manipulating Widgets”](#) on page 26 for details on using `WIDGET_CONTROL`.) Between the time when the widget is created as an IDL widget record and when it is realized as a platform-specific interface element, you have control over many, but not all, aspects of the widget's state. Some details of the final realized widget's state (such as its exact screen geometry) may remain undetermined until the widget is instantiated. Realization, and the related concepts of *mapping* and *sensitivity*, are discussed in greater in following sections.

It is important to note that unrealized widgets in a widget hierarchy can be manipulated programmatically. Examples of attributes you can manipulate before realization are the overall geometry of the user interface, widget values, and user values. You can even retrieve widget values before the widgets are realized. Unrealized widgets do not, however, generate widget events, since the actual platform-specific user interface has yet to be created.

Once a widget has been realized, its corresponding platform-specific user interface toolkit element is instantiated. The native toolkit determines the widget's exact screen geometry. If the widget is then *mapped*, it becomes visible on the computer screen, can be manipulated by a user, and generates widget events.

Note

Widgets are mapped by default. This means that when you realize a widget hierarchy, the widgets included in that hierarchy will usually be displayed on screen immediately. You can control the visibility of widget hierarchies — before or after realization — using the `MAP` keyword to `WIDGET_CONTROL`. See [“Controlling Widget Visibility”](#) on page 27 for details.

Note also that widgets that are visible on screen can be made unavailable to the user by setting the `SENSITIVE` keyword to `WIDGET_CONTROL`. See [“Sensitizing Widgets”](#) on page 28 for details.

Example: A Simple Widget Application

The following example demonstrates the simplicity of widget programming. The example program creates a base widget containing a single button, labelled “Done.” When you position the mouse cursor over the button and click, the widget is destroyed.

Note

If you are new to IDL widget programming, don't be dismayed if parts of this example are not immediately clear to you. As you read further through this chapter, the principles of the event-driven programming model and IDL's specific implementation of that model will become clearer.

Example Code

This example is included in the file `doc_widget1.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution. Run this example procedure by entering `doc_widget1` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT doc_widget1.pro`. See [“Running the Example Code”](#) on page 15 if IDL does not run the program as expected.

```
PRO doc_widget1_event, ev
  IF ev.SELECT THEN WIDGET_CONTROL, ev.TOP, /DESTROY
END

PRO doc_widget1
  base = WIDGET_BASE(/COLUMN)
  button = WIDGET_BUTTON(base, value='Done')
  WIDGET_CONTROL, base, /REALIZE
  XMANAGER, 'doc_widget1', base
END
```

While this simple example does nothing particularly useful, it does illustrate some basic concepts of event-driven programming. Let's examine how the example is constructed.

First, note that the “application” consists of two parts: an event handling routine and a creation routine. Let's first examine the second part — the creation routine — contained in the `doc_widget1` procedure.

The `doc_widget1` procedure does the following:

1. Creates a top-level base widget whose widget ID is stored in the variable `base`. All widget applications have at least one base.
2. Creates a button widget whose widget ID is stored in the variable `button`. The button widget has `base` as its parent. The value “Done” is assigned to the button. The value of a button widget is the text that appears on the button’s face.
3. Realizes the widget hierarchy built on `base` by calling `WIDGET_CONTROL` with the `/REALIZE` keyword. Realizing the widget hierarchy displays the widget on your computer screen.
4. Invokes the `XMANAGER` routine to manage the widget event loop, providing the name of the calling routine (`doc_widget1`) and the widget ID of the top-level base on which the widget hierarchy is built (`base`).

The `doc_widget1_event` procedure is the event handling routine for the application. By convention, the `XMANAGER` procedure looks for an event handling procedure with the same name as the procedure that creates the widgets, with “_event” appended to the end. (This default can be overridden by specifying an event handler directly using the `EVENT_HANDLER` keyword to `XMANAGER`.) When an event is received by `XMANAGER`, the event structure is passed to the `doc_widget1_event` procedure via the `ev` argument.

In this example, all the event handling routine does is check the event structure to see if the event passed to it was a select event generated by the button widget. If a `SELECT` event is received, the routine calls `WIDGET_CONTROL` with the `DESTROY` keyword to destroy the widget hierarchy built on the top-level base widget (specified in the `TOP` field of the event structure).

For further discussion of widget events and event structures, see “[Widget Event Processing](#)” on page 34. For details about the event structures returned by different widgets, see the documentation for each widget in the *IDL Reference Guide*.

Widget Application Lifecycle

When you create and use a widget application, you do the following things:

1. [Construct the Widget Hierarchy](#)
2. [Provide an Event-Handling Routine](#)
3. [Realize the Widgets](#)
4. [Register the Program with the XMANAGER](#)
5. [Interact with the Application](#)
6. [Destroy the Widgets](#)

Construct the Widget Hierarchy

You must first build a widget hierarchy using the `WIDGET_*` functions. Start by creating a *top-level base* with the `WIDGET_BASE` function.

Combine other widget creation functions — `WIDGET_BUTTON`, `CW_PDMENU`, etc. — to create and organize the user interface of your widget application. At this point, the widgets are *unrealized* — they exist only as IDL widget records — and nothing has been created or displayed on the screen.

Note

Widget applications can include multiple widget hierarchies headed by multiple top-level base widgets. See [“Using Multiple Widget Hierarchies”](#) on page 61 for more on creating a hierarchy of widget hierarchies.

Provide an Event-Handling Routine

In order for a widget application to *do* anything, you must provide a routine that examines events, determines what action to take, and implements that action. Actions may involve computation, graphics display, or updates to the widget interface itself.

For best performance, event processing routines must run and return to the calling routine as quickly as possible. Widgets won't respond to user input while the event-processing routine is running. Widget-based programs should wait for user-generated events, handle them as quickly as possible, and return to wait for more events. Event processing is discussed in detail in [“Widget Event Processing”](#) on page 34.

Event handling routines can manipulate widgets via the `WIDGET_CONTROL` procedure. Possible actions include the following:

- Obtain or change the value of a widget (see “[Widget Values](#)” on page 18) using the `APPEND`, `GET_VALUE`, and `SET_VALUE` keywords.
- Obtain or change the value of a widget’s user value using the `GET_UVALUE` and `SET_UVALUE` keywords. (User values are discussed in “[Widget User Values](#)” on page 33)
- Map and unmap widgets using the `MAP` keyword. Unmapped widgets are removed from the screen and become invisible, but they still exist in memory.
- Change a widget’s sensitivity using the `SENSITIVE` keyword. A widget indicates that it is insensitive by changing its appearance (often by graying itself or displaying text with dashed lines) and ignoring any user input. It is useful to make widgets insensitive at points where it would be inconvenient to get events from them (for example, if your program is waiting for input from another source).
- Change the settings of toggle buttons using the `SET_BUTTON` keyword.
- Push a widget hierarchy behind the other windows on the screen, or pull it in front, using the `SHOW` keyword.
- Display the “hourglass” cursor while the application is busy and not able to respond to user actions by setting the `HOURGLASS` keyword. (See “[Indicating Time-Consuming Operations](#)” on page 28.)

Realize the Widgets

To convert the IDL widget records representing your widget hierarchy into a set of platform-specific user interface toolkit elements, use the `REALIZE` keyword to the `WIDGET_CONTROL` procedure. Unless you have specifically *unmapped* the widgets before realizing them, the `REALIZE` keyword causes the widgets to be displayed on screen. See “[Manipulating Widgets](#)” on page 26 for additional details.

Register the Program with the XMANAGER

Your widget application waits for events to be reported to it and reacts as specified in the event handling routine after being registered with the `XMANAGER` procedure.

Events are obtained by `XMANAGER` via the `WIDGET_EVENT` function and passed to the calling routine (your event handler) in the form of an IDL structure variable. Each type of widget returns a different type of structure, as described in the documentation for the individual widget creation functions in the *IDL Reference*

Guide. Every event structure has three common elements: long integers named `ID`, `TOP`, and `HANDLER`:

- `ID` is the widget ID of the widget generating the event.
- `TOP` is the widget ID of the top-level base containing the widget that generated the event.
- `HANDLER` is important for event handler functions, which are discussed later in this chapter.

When an event occurs, `XMANAGER` arranges for the event structure to be passed to an event-handling procedure specified by the program, and the event handler takes some appropriate action based on the event. This means that multiple widget applications can run simultaneously — `XMANAGER` arranges for the events be dispatched to the appropriate routine.

Interact with the Application

Once the widget application has been realized and registered with `XMANAGER`, the user can interact with the application to accomplish whatever tasks the application is designed to accomplish.

Destroy the Widgets

When the application has finished (usually when the user clicks on a “Done” or “Quit” button), destroy the widget hierarchy using the `DESTROY` keyword to the `WIDGET_CONTROL` procedure. This causes all resources related to the hierarchy to be freed and removes it from the screen.

Manipulating Widgets

IDL provides several routines that allow you to manipulate and manage widgets programmatically:

- **WIDGET_CONTROL** allows you to realize widget hierarchies, manipulate them, and destroy them.
- **WIDGET_EVENT** allows you to process events generated by a specific widget hierarchy.
- **WIDGET_INFO** allows you to obtain information about the state of a specific widget or widget hierarchy.
- **XMANAGER** provides an event loop and manages events generated by all existing widget hierarchies.
- **XREGISTERED** allows you to test whether a specific widget is currently registered with XMANAGER.

These widget manipulation routines are discussed in more detail in the following sections.

WIDGET_CONTROL

The **WIDGET_CONTROL** procedure allows you to realize, manage, and destroy widget hierarchies. It is often used to change the default behavior or appearance of previously-realized widgets.

Keywords to **WIDGET_CONTROL** may affect only certain types of widgets, any type of widget, or the widget system in general. See “**WIDGET_CONTROL**” (*IDL Reference Guide*) for complete details. We discuss here only a few of the more common uses of this procedure.

Realizing Widget Hierarchies

IDL widgets are actually *widget records* that represent platform-specific user interface toolkit elements. In order to instantiate the platform-specific toolkit elements, widgets must be *realized* with the following statement:

```
WIDGET_CONTROL, base, /REALIZE
```

where `base` is the widget ID of the top-level base widget for your widget hierarchy.

Destroying Widget Hierarchies

The standard way to destroy a widget hierarchy is with the statement:

```
WIDGET_CONTROL, base, /DESTROY
```

where `base` is the widget ID of the top-level base widget of the hierarchy to be killed. Usually, IDL programs that use widgets issue this statement in their event-handling routine in response to the user's clicking on a "Done" button in the application.

In addition, some window managers place a pulldown menu on the frame of the top-level base widget that allows the user to kill the entire hierarchy. Using the window manager to kill a widget hierarchy is equivalent to using the `DESTROY` keyword to the `WIDGET_CONTROL` procedure.

When designing widget applications, you should always include a "Done" button (or some other widget that allows the user to exit) in the application itself, since some window managers do not provide the user with a kill option from the outer frame.

Retrieving or Changing Widget Values

You can use `WIDGET_CONTROL` to retrieve or change widget values using the `GET_VALUE` and `SET_VALUE` keywords. Similarly, you can retrieve or change widget user values with the `GET_UVALUE` and `SET_UVALUE` keywords.

For example, you could use the following commands to retrieve the value of a draw widget whose widget ID is stored in the variable `drawwid`, and to make that draw widget the current graphics window:

```
WIDGET_CONTROL, drawwid, GET_VALUE=draw  
WSET, draw
```

Similarly, you could use the following command in an event handling procedure to save the user value of the widget that generates an event into an IDL variable named `uval`:

```
WIDGET_CONTROL, event.id, GET_UVALUE=uval
```

For more on widget user values, see ["Widget User Values"](#) on page 33.

Controlling Widget Visibility

You can display or remove realized widgets from the screen by *mapping* or *unmapping* them. Unmapped widgets still exist in the widget hierarchy, but they are not displayed and do not generate events.

Set the MAP keyword to WIDGET_CONTROL equal to zero to hide a widget, or to a nonzero value to display it again. For example, to hide the `base1` widget and all its child widgets from view, use the following command:

```
WIDGET_CONTROL, base1, MAP=0
```

By default, widgets are mapped automatically when they are realized. You can prevent a widget from appearing on screen when you realize it by setting MAP=0 *before* realizing the widget hierarchy.

Note

While it is possible to call WIDGET_CONTROL, MAP=0 with the widget ID of any widget, only base widgets can actually be unmapped. If you specify a widget ID that is not from a base widget, IDL searches upward in the widget hierarchy until it finds the closest base widget. The map operation is applied to that base.

Sensitizing Widgets

Use sensitivity to control when a user is allowed to manipulate a widget. When a widget is sensitive, it has a normal appearance and can receive user input. When a widget is insensitive, it ignores any input directed at it. Note that while most widgets change their appearance when they become insensitive, some simply stop generating events.

Set the SENSITIVE keyword equal to zero to desensitize a widget, or to a nonzero value to make it sensitive. For example, you might wish to make a group of buttons contained in a base whose widget ID is stored in the variable `bgroup` insensitive after some user input. You would use the following command:

```
WIDGET_CONTROL, bgroup, SENSITIVE=0
```

Indicating Time-Consuming Operations

In an event driven environment, it is important that the interface be highly responsive to the user's manipulations. Widget event handlers should be written to execute quickly and return. However, sometimes the event handler has no option but to perform an operation that is slow. In such a case, it is a good idea to give the user feedback that the system is busy. This is easily done using the HOURGLASS keyword just before the expensive operation is started:

```
WIDGET_CONTROL, /HOURGLASS
```

This command causes IDL to turn on an hourglass-shaped cursor for all IDL widgets and graphics windows. The hourglass remains active until the next event is processed, at which point the previous cursor is automatically restored.

WIDGET_EVENT

The `WIDGET_EVENT` function returns events for the widget hierarchy rooted at `Widget_ID`. Events are generated when a button is pressed, a slider position is changed, and so forth. In most cases, you will not use `WIDGET_EVENT` directly, but instead will use the `XMANAGER` routine to manage widget events. Event processing is discussed in detail in “[Widget Event Processing](#)” on page 34. See also “[WIDGET_EVENT](#)” (*IDL Reference Guide*) for additional details.

WIDGET_INFO

The `WIDGET_INFO` function is used to obtain information about the widget subsystem and individual widgets. You supply the widget ID of a widget for which you want to retrieve some information, along with a keyword that specifies the type of information. For example, to determine the index of the selected item in a list widget whose widget ID is contained in the variable `list`, you would use a command like the following:

```
listindex = WIDGET_INFO(list, /LIST_SELECT)
```

Finding Widget IDs using WIDGET_INFO

One noteworthy use of `WIDGET_INFO` is to locate the widget ID of a widget with a specified *user name*. (A *user name* is a part of the widget’s widget record that contains a text identifier, specified by the programmer.) See “[Working With Widget IDs](#)” on page 31 for more information on this technique.

See “[WIDGET_INFO](#)” (*IDL Reference Guide*) for more information.

XMANAGER

The `XMANAGER` procedure provides the main event loop registration and widget management. Calling `XMANAGER` “registers” a widget program with the `XMANAGER` event handler. `XMANAGER` takes control of event processing until all widgets have been destroyed.

Using `XMANAGER` allows you to run multiple widget applications and work at the IDL command line at the same time. While it is possible to use `WIDGET_EVENT` directly to manage events in your application, it is almost always easier to use `XMANAGER`.

See “[XMANAGER](#)” (*IDL Reference Guide*) for complete details.

XREGISTERED

The XREGISTERED function returns True if the widget specified by its argument is currently registered with the XMANAGER.

One use of the XREGISTERED function is to control the number of instances of a given widget application that run at a given time. For example, suppose that you have a widget program that registers itself with the XMANAGER with the command:

```
XMANAGER, 'mywidget', base
```

You could limit this widget to one instantiation by adding the following line as the first line (after the procedure definition statement) of the widget creation routine:

```
IF (XREGISTERED('mywidget') NE 0) THEN RETURN
```

See “[XREGISTERED](#)” (*IDL Reference Guide*) for complete details.

Working With Widget IDs

Any widget application capable of doing real work will include one or more routines that are separate from the routine that creates the widget hierarchy, designed to handle and respond to user-generated events. *Event processing routines* — the routines that process information contained in widget event structures and respond accordingly — often retrieve information contained in the widget values of the widgets that make up the interface, perform calculations, and modify the widget interface itself in response to user actions.

Since a widget ID is required to retrieve information from or set values in a widget, you will need a way for your event processing routines to retrieve the ID of a specified widget. This section describes techniques you can use to pass widget IDs between the routines in your widget application — most notably between the widget creation routine (where widget IDs are generated) and the event processing routines.

Use the Widget Event Structure

Every time a user interacts with a widget using the mouse or keyboard, a *widget event structure* is generated. Widget event structures contain the widget ID of the widget that generated the event. In addition, widget event structures provide the widget ID of the top-level base in the widget hierarchy to which the widget the generated the event belongs.

Getting the widget ID of the appropriate widget from the event structure is almost always the preferred method for passing a widget ID from one routine to another within your application. Widget event processing is discussed in detail in [“Widget Event Processing”](#) on page 34.

Pass the Widget ID Using a Widget User Value

The widget event structure always includes two widget IDs: the ID of the widget that generated the event, and the ID of the top-level base widget. If you need to pass multiple widget IDs between routines, it is often useful to place the widget ID values in the *user value* of the top-level base widget. Widget user values are discussed in [“Widget User Values”](#) on page 33.

Use a User Name to Locate the Widget

One of the pieces of information you can specify when you create a widget is a *user name*. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID. To specify

a user name, set the UNAME keyword to the widget creation routine equal to a string that can be used to identify the widget in your code.

To query the widget hierarchy, use the `WIDGET_INFO` function with the widget ID of the top-level base widget and the `FIND_BY_UNAME` keyword. Note that user names must be unique within the widget hierarchy, because the `FIND_BY_UNAME` keyword returns the ID of the first widget with the specified name.

Pass the Widget ID Explicitly

In some cases, you may need to pass a specific widget ID available in one routine to a second routine. In this case, you can specify the widget ID as a parameter when calling the second routine from the first. While this method is not so general as using the widget event structure, it is useful in some circumstances.

Use a COMMON Block

In rare cases, it may be useful to store widget IDs in a `COMMON` block, making them available to all routines in the application. While using a `COMMON` block may seem like a good strategy on first inspection, this method has several drawbacks. Most importantly, using a `COMMON` block to hold widget IDs means that only one instance of a given widget application can be running at once.

Widget User Values

Every widget primitive and compound widget can carry a user-specified value of any IDL data type and organization; that is, every widget contains a variable that can store arbitrary information. This value is ignored by the widget and is for the programmer's convenience only.

The initial *user value* is specified using the UVALUE keyword to the widget creation function. If no initial value is specified, the user value is undefined. Once the widget exists, its user value can be examined and/or changed using the GET_UVALUE and SET_UVALUE keywords to the WIDGET_CONTROL procedure.

Note

The *widget user value* should not be confused with the *widget value*, described in “[Widget Values](#)” on page 18.

User Values Simplify Event Handling

User values can be used to simplify event-handling. If each widget has a distinct user value, you need only check the user value of any event to determine which widget generated it. In practice, this means you do not need to keep track of the widget IDs of all the widgets in your widget hierarchy in order to determine what to do with a given event.

User Values can be Accessible Throughout a Widget Application

Another use for user variables is to simulate a variable that is available in more than one IDL routine. For example, you can set the user value of a top-level base widget equal to one or more widget IDs. You then have an easy way to pass the widget IDs from your widget creation routine to your event handling routine.

We will take advantage of both of these aspects of user values in “[Example: Event Processing and User Values](#)” on page 40.

Widget Event Processing

The concepts of events and event processing underlie every aspect of widget programming. It is important to understand how IDL handles widget events in order to use widgets effectively.

This section discusses the following topics:

- [What are Widget Events?](#)
- [Structure of Widget Events](#)
- [Managing Widget Events with XMANAGER](#)
- [Event Processing and Callbacks](#)

For a discussion of techniques you can use to detect and respond to specific types of events, see “[Working with Widget Events](#)” in Chapter 3.

What are Widget Events?

A widget event is a message returned from the window system when a user manipulates a widget. In response to an event, a widget program usually performs some action (*e.g.*, opens a file, updates a plot).

Structure of Widget Events

As events arrive from the window system, IDL saves them in a queue for the target widget. The `WIDGET_EVENT` function delivers these events to the IDL program as IDL structures. Every widget event structure has the same first three fields: these are long integers named `ID`, `TOP`, and `HANDLER`:

- `ID` is the widget ID of the widget that generated the event.
- `TOP` is the widget ID of the top-level base containing `ID`.
- `HANDLER` is the widget ID of the widget associated with the event handling routine. The importance of `HANDLER` will become apparent when we discuss event routines and compound widgets, below.

Event structures for different widgets may contain other fields as well. The exact form of the event structure for any given widget is described in the documentation for that widget’s creation function in the *IDL Reference Guide*.

Managing Widget Events with XMANAGER

The XMANAGER procedure provides a convenient, simplified interface IDL's event-handling capabilities. At the highest level, creating a widget application consists of the following steps:

1. Creating routines to react to widget events.
2. Creating the widgets that make up the application's interface.
3. Realizing the widgets.
4. Calling XMANAGER to manage events flowing from the widget interface.

XMANAGER arranges for an event-handling procedure supplied by the application to be called when events for it arrive. The application is shielded from the details of calling the underlying WIDGET_EVENT function and interacting with other widget applications that may be running simultaneously.

Note

While it is possible for a user-written program to call the WIDGET_EVENT function directly, in practice this is very unusual. For details on how events are handled at a low level, see [“The WIDGET_EVENT Function”](#) on page 37.

The file `xmng_tmpl.pro`, found in the `lib` subdirectory of the IDL distribution, is a template for writing widget applications that use XMANAGER.

XMANAGER and *Blocking*

The term *blocking* is used to describe a situation in which processing by IDL is suspended until some event or action takes place. Unless you specifically arrange otherwise, IDL will only allow one user interface (the IDL command line or a single widget application) to be active at one time. XMANAGER simplifies the process of arranging things so that multiple user interfaces can run at the same time — that is, managing events so that applications do not need to *block* in order to be assured of receiving the correct event information.

IDL's blocking behavior is discussed in detail in [“XMANAGER”](#) (*IDL Reference Guide*). In most cases, specifying the `NO_BLOCK` keyword when calling XMANAGER will allow your application to “play nicely with others,” but you should keep the following things in mind when writing widget applications:

Active Command Line

IDL can provide an *active command line*. If the command line is active, IDL will execute commands entered at the command line even if one or more widget

applications are already running. In order for IDL to behave in this way, all widget applications must be run via XMANAGER with the NO_BLOCK keyword set. See “[Active Command Line](#)” under “XMANAGER” (*IDL Reference Guide*) for details

Blocking and Non-Blocking Applications

By default, widget applications — even those managed with XMANAGER — will block. To enable your application to run without blocking other widget applications or the IDL command line, you must explicitly set the NO_BLOCK keyword to XMANAGER when registering the application. Put another way, any running widget application that does not have this keyword set will block all event processing for widget applications and the IDL command line. See “[Blocking vs. Non-blocking Applications](#)” under “XMANAGER” (*IDL Reference Guide*) for details.

Registering Applications Without Processing Their Events

In order to allow multiple widget applications to run simultaneously, each application must be *registered* with XMANAGER, so it knows how to recognize events generated by the application. In most cases, the registration step takes place automatically when XMANAGER is called to begin processing events for the application.

In some cases, however, it may be useful to register an application with XMANAGER before asking it to begin processing the application’s events. In these cases, you can use the JUST_REG keyword to XMANAGER; the application is added to XMANAGER’s list of known applications without starting event processing, and XMANAGER returns immediately. See “[JUST_REG vs. NO_BLOCK](#)” under “XMANAGER” (*IDL Reference Guide*) for details.

Tips on Working With XMANAGER

Because XMANAGER buffers you from direct handling of widget events, you *cannot* explicitly specify an event-handling function or procedure for the top-level base using the EVENT_FUNC or EVENT_PRO keywords to WIDGET_BASE or WIDGET_CONTROL. Event handlers for top-level bases specified via these keywords will be overwritten by XMANAGER.

Instead, provide the name of the event handler routine to XMANAGER via the EVENT_HANDLER keyword. If you do not supply the name of an event handler via the EVENT_HANDLER keyword, XMANAGER will construct a default name by adding the suffix “_event” to the *Name* argument.

Note that this guideline applies only to top-level bases (base widgets created with no parent widget). Child base widgets should use the EVENT_FUNC or EVENT_PRO keywords to specify event handling routines, if necessary.

In addition, it is often convenient to specify the death-notification routine for the top-level base of a widget application via the CLEANUP routine to XMANAGER rather than via the KILL_NOTIFY keyword to WIDGET_BASE or WIDGET_CONTROL. Either method will work, but the *last* routine specified is the routine that will be called when the base widget is destroyed. Since the call to XMANAGER is often the last call made when creating a widget application, using the CLEANUP keyword to specify the routine to be called when the application ends is preferred.

The XREGISTERED Function

The XMANAGER procedure allows multiple instances of a widget application to run simultaneously. In some cases, however, you may wish to ensure that only a single instance of application can run at a given time. An obvious example of this is an application that uses a COMMON block to maintain its current state (see “[Managing Application State](#)” on page 42).

The XREGISTERED function can be used in such applications to ensure that only a single copy of the application run at a time. Place the following statement at the start of the widget creation routine:

```
IF (XREGISTERED('routine_name') NE 0) THEN RETURN
```

where *routine_name* is the name of the widget application.

See “[XREGISTERED](#)” (*IDL Reference Guide*) for further information.

The WIDGET_EVENT Function

All widget event processing in IDL is eventually handled by the [WIDGET_EVENT](#) function. Note that while we will discuss WIDGET_EVENT here for completeness, in most cases you will *not* want to call WIDGET_EVENT directly. The [XMANAGER](#) routine provides a convenient, simplified interface to WIDGET_EVENT and allows IDL to take over the task of managing multiple widget applications.

In its simplest form, the WIDGET_EVENT function is called with a widget ID (usually, the ID of a base widget) as its argument. WIDGET_EVENT checks the queue of undelivered events for that widget *or any of its children*. If an event is present, it is immediately dequeued and returned. If no event is available, WIDGET_EVENT blocks all other processing by IDL until an event arrives, and then returns it. Typically, the request is made for a top-level base, so WIDGET_EVENT returns events for any widget in the widget hierarchy rooted at that base widget.

This simple usage suffers from a major weakness. Since each call to WIDGET_EVENT is looking for events from a specified widget hierarchy, it is not possible to receive events for more than one widget hierarchy at a time. It is important

to be able to run multiple widget applications (each with a separate top-level base) simultaneously. An example would be an image processing application, a color table manipulation tool, and an on-line help reader all running together.

One solution to this problem is to call `WIDGET_EVENT` with an array of widget identifiers instead of a single ID. In this case, `WIDGET_EVENT` returns events for any widget hierarchy in the list. This solution is effective, but it still requires that you maintain a complete list of all interesting top-level base identifiers, which implies that all cooperating applications need to know about each other.

The most powerful way to use `WIDGET_EVENT` is to call it without any arguments at all. Called this way, it will return events for any currently-realized widgets that have expressed an interest in being managed. (You specify that a widget wants to be managed by setting the `MANAGED` keyword to the `WIDGET_CONTROL` procedure.) This form of `WIDGET_EVENT` is especially useful when used in conjunction with widget event callback routines, discussed in “[Event Processing and Callbacks](#)” on page 38.

Event Processing and Callbacks

Previously, we mentioned that when IDL receives an event, the event is queued until a call to `WIDGET_EVENT` is made (either explicitly by the user program or by `XMANAGER`), whereupon the event is dequeued and returned. The following is a more complete description of what actually happens in IDL’s *event loop*.

Events for a given widget are processed in the order that they are generated. The event processing performed by `WIDGET_EVENT` consists of the following steps, applied iteratively:

1. Wait for an event from one of the specified widgets to arrive.
2. Starting with the widget that generated the event, search up the widget hierarchy for a widget with an associated event-handling procedure or function.

Event-handling routines associated with widgets are known as *callback* routines. Other cases where an IDL system routine (`WIDGET_EVENT`, in this instance) calls a user-specified, user-written routine include routines specified via the `KILL_NOTIFY` or `NOTIFY_REALIZE` keywords to the widget creation functions and `WIDGET_CONTROL`, as well as the corollary keywords to `XMANAGER`.

3. If an event-handling *procedure* is found, it is called with the event as its argument. The `HANDLER` field of the event is set to the widget ID of the widget associated with the handling procedure. When the procedure returns,

WIDGET_EVENT returns to the first step above and starts searching for events. Hence, event-handling procedures are said to “swallow” events.

4. If an event-handling *function* is found, it is called with the event as its argument. The HANDLER field of the event is set to the widget ID of the widget associated with the handling function.

When the function returns, its value is examined. If the value is not a structure, it is discarded and WIDGET_EVENT returns to the first step. This behavior allows event-handling functions to selectively act like event-handling procedures and “swallow” events.

If the returned value is a structure, it is checked to ensure that it has the standard first three fields: ID, TOP, and HANDLER. If any of these fields is missing, IDL issues an error. Otherwise, the returned value replaces the event found in the first step and WIDGET_EVENT continues moving up the widget hierarchy looking for another event handler routine, as described in step 2, above.

In situations where an event structure is returned, event functions are said to “rewrite” events. This ability to rewrite events is the basis of *compound widgets*, which combine several widgets to give the appearance of a single, more complicated widget. Compound widgets are an important widget programming concept. For more information, see [“Creating a Compound Widget”](#) on page 46.

5. If an event reaches the top of a widget hierarchy without being swallowed by an event handler, it is returned as the value of WIDGET_EVENT.
6. If WIDGET_EVENT was called without an argument, and there are no widgets left on the screen that are being managed (as specified via the MANAGED keyword to the WIDGET_CONTROL procedure) and could generate events, WIDGET_EVENT ends the search and returns an *empty event* (a standard widget event structure with the top three fields set to zero).

Example: Event Processing and User Values

The following example demonstrates how user values can be used to simplify event processing and to pass values between routines. It creates a base widget with three buttons and a text field that reports which button was pressed.

Note

If you are new to IDL widget programming, don't be worried if parts of this example are not immediately clear to you. As you read further through this chapter, the principles of the event-driven programming model and IDL's specific implementation of that model will become clearer.

Example Code

This example is included in the file `doc_widget2.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution. Run this example procedure by entering `doc_widget2` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT doc_widget2.pro`. See [“Running the Example Code”](#) on page 15 if IDL does not run the program as expected.

```

PRO doc_widget2_event, ev
  WIDGET_CONTROL, ev.TOP, GET_UVALUE=textwid
  WIDGET_CONTROL, ev.ID, GET_UVALUE=uval
  CASE uval OF
    'ONE' : WIDGET_CONTROL, textwid, SET_VALUE='Button 1 Pressed'
    'TWO' : WIDGET_CONTROL, textwid, SET_VALUE='Button 2 Pressed'
    'DONE': WIDGET_CONTROL, ev.TOP, /DESTROY
  ENDCASE
END

PRO doc_widget2
  base = WIDGET_BASE(/COLUMN)
  button1 = WIDGET_BUTTON(base, VALUE='One', UVALUE='ONE')
  button2 = WIDGET_BUTTON(base, VALUE='Two', UVALUE='TWO')
  text = WIDGET_TEXT(base, XSIZE=20)
  button3 = WIDGET_BUTTON(base, value='Done', UVALUE='DONE')
  WIDGET_CONTROL, base, SET_UVALUE=text
  WIDGET_CONTROL, base, /REALIZE
  XMANAGER, 'doc_widget2', base
END

```

Let's examine the creation routine, `doc_widget2`, first. We first create a top-level base, this time specifying the `COLUMN` keyword to ensure that the widgets contained in the base are stacked vertically. We create two buttons with values “One” and “Two,” and user values “ONE” and “TWO.” Remember that the *value* of a button widget is also the button's label. We create a text widget, and specify its width to be

20 characters using the `XSIZE` keyword. The last button is the “Done” button, with a the user value “DONE.”

Next follow two calls to the `WIDGET_CONTROL` procedure. The first call sets the user value of the top-level base widget equal to the widget ID of our text widget, allowing easy access to the text widget from the event handling routine. The second call realizes the top-level base and all its child widgets. Finally, we invoke the `XMANAGER` to manage the widget application.

The `doc_widget2_event` routine is slightly more complicated than the event handler in “[Example: A Simple Widget Application](#)” on page 21, but it is still relatively simple. We begin by using `WIDGET_CONTROL` to retrieve the widget ID of our text widget from the user value of the top-level base. We can do this because the widget ID of our top-level base is contained in the `TOP` field of the widget event structure. We use the `GET_UVALUE` keyword to store the widget ID of the text widget in the variable `textwid`.

Next, we use `WIDGET_CONTROL` with the `GET_UVALUE` keyword to retrieve the user value of the widget that generated the event. Again, we can do this because we know that the widget ID of the widget that generated the event is stored in the `ID` field of the event structure. We then use a `CASE` statement to compare the user value of the widget, now stored in the variable `uval`, with the list of possible user values to determine which button was pressed and act accordingly.

In the `CASE` statement, we check to see if `uval` is the user value associated with either button one or button two. If it is, we use `WIDGET_CONTROL` and the `SET_VALUE` keyword to alter the value of the text widget, whose ID we stored in the variable `textwid`. If `uval` is 'DONE', we recognize that the user has clicked on the “Done” button and use `WIDGET_CONTROL` to destroy the widget hierarchy.

Managing Application State

A widget application is usually divided into at least two separate routines, one that creates and realizes the application and another that handles events. These multiple routines need shared access to certain types of information, such as the widget IDs of the application's widgets and data being used by the application. This shared information is referred to as the *application state*.

Techniques for Preserving Application State

The following are some techniques you can use to preserve and share application state data between routines.

Using COMMON Blocks

One obvious answer to this problem is to use a COMMON block to hold the state. However, this solution is generally undesirable because it prevents more than a single copy of the application from running at the same time. It is easy to imagine the chaos that would ensue if multiple instances of the same application were using the *same* common block without some sort of interlocking.

Using a State Structure in a User Value

A better solution to this problem is to use the user value of one of the widgets to store state information for the application. Using this technique, multiple instances of the same widget code can exist simultaneously. Since this user value can be of any type, a structure can be used to store any number of state-related values.

For example, consider the following example widget code:

```

PRO my_widget_event, event
    WIDGET_CONTROL, event.TOP, GET_UVALUE=state, /NO_COPY

    Event-handling code goes here

    WIDGET_CONTROL, event.TOP, SET_UVALUE=state, /NO_COPY
END

PRO my_widget
; Create some widgets
wBase = WIDGET_BASE(/COLUMN)
wDraw = WIDGET_DRAW(wBase, XSIZE=300, YSIZE=300)

; Realize the base widget and retrieve the widget ID
; of the drawable area.
```

```

WIDGET_CONTROL, wBase, /REALIZE
WIDGET_CONTROL, wDraw, GET_VALUE=idxDraw

; Create a state structure variable and set the user
; value of the top-level base equal to the state variable.
state = {wDraw:wDraw, idxDraw:idxDraw}
WIDGET_CONTROL, wBase, SET_UVALUE=state

; Use XMANAGER to manage the widgets
XMANAGER, 'my_widget', wBase
END

```

In this example, we store state information (the widget ID of the draw widget and the index of the drawable area) in a structure variable, and set the user value of the top-level base widget equal to that structure variable. This makes it possible to retrieve the structure using the widget ID contained in the TOP field of any widget event structure that arrives at the event handler routine.

Notice the use of the NO_COPY keyword to WIDGET_CONTROL in the example. This keyword prevents IDL from duplicating the memory used by the user value during the GET_UVALUE and SET_UVALUE operations. This is an important efficiency consideration if the size of the state data is large. (In this example the use of NO_COPY is not really necessary, as the state data consists only of the two long integers that represent the widget IDs being passed in the state variable.)

While it is important to consider efficiency, the use of the NO_COPY keyword does have the side effect of causing the user value of the widget to become undefined when it is retrieved using the GET_UVALUE keyword. If the user value is not replaced before the event handler exits, the next execution of the event routine will fail, since the user value will be undefined.

Using a Pointer to the State Structure

A variation on the above technique uses an IDL pointer to contain the state variable. This eliminates the duplication of data and the need for the use of the NO_COPY keyword.

Consider the following example widget code:

```

PRO my_widget_event, event
  WIDGET_CONTROL, event.TOP, GET_UVALUE=pState

  Event-handling code goes here, accessing the state
  structure via the retrieved pointer.

END

PRO my_widget_cleanup, wBase

```

```

; This routine is called when the application quits.
; Retrieve the state variable and free the pointer.
WIDGET_CONTROL, wBase, GET_UVALUE=pState
PTR_FREE, pState
END

PRO my_widget
; Create some widgets.
wBase = WIDGET_BASE(/COLUMN)
wDraw = WIDGET_DRAW(wBase, XSIZE=300, YSIZE=300)

; Realize the base widget and retrieve the widget ID
; of the drawable area.
WIDGET_CONTROL, wBase, /REALIZE
WIDGET_CONTROL, wDraw, GET_VALUE=idxDraw

; Create a state structure variable.
state = {wDraw:wDraw, idxDraw:idxDraw}

; Place the state structure in a pointer and set the user
; value of the top-level base widget equal to the pointer.
pState = PTR_NEW(state, /NO_COPY)
WIDGET_CONTROL, wBase, SET_UVALUE=pState, /NO_COPY

; Call XMANAGER to manage the widgets, specifying the routine
; to be called when the application quits.
XMANAGER, 'my_widget', wBase, CLEANUP='my_widget_cleanup'
END

```

Notice the following differences between this technique and the technique shown in the previous example:

- This method eliminates the removal of the user value from the top-level base widget by removing the use of the `NO_COPY` keyword with the `GET_UVALUE` keyword to `WIDGET_CONTROL`. Since only the pointer (a long integer) is passed to the event routine, the efficiency issues connected with copying the value are small enough to ignore. (Note that we do use the `NO_COPY` keyword when creating the pointer and when initially setting the user value of the top-level base widget; since these statements are executed only once, we don't worry about the fact that the `state` or `pState` variables become undefined.)
- The state structure contained in the pointer must now be referenced using pointer-dereferencing syntax. For example, to refer to the `idxDraw` field of the state structure within the event-handling routine, you would use the syntax

```
(*pState).idxDraw
```

- The pointer allocated to store the state structure must be freed when the widget application quits. We do this by specifying a cleanup routine via the `CLEANUP` keyword to `XMANAGER`. It is the cleanup routine's responsibility to free the pointer.

Each of the above techniques has advantages. Choose a method based on the complexity of your application and your level of comfort with features like IDL pointers and the `NO_COPY` keyword.

Creating a Compound Widget

Widget primitives can be used to construct many varied user interfaces, but complex programs written with them suffer the following drawbacks:

- Large widget applications become difficult to maintain. As an application grows, it becomes more difficult to properly write and test. The resulting program suffers from poor organization.
- Good ideas can be difficult to reuse. Most larger applications are constructed from smaller sub-units. For example, a color table editor might contain control panel, color selection and color-index selection sub-units. These sub-units are often complicated tools that could be used profitably in other programs. To reuse such sub-units, the programmer must understand the existing application and then transplant the interesting parts into the new program — at best a tedious and error-prone proposition.

Compound widgets solve these problems. A compound widget is a complete, self-contained, reusable widget sub-tree that behaves to a large degree just like a primitive widget. Complex widget applications written with compound widgets are much easier to maintain than the same application written without them. Using compound widgets is analogous to using subroutines and functions in programming languages.

Writing Compound Widgets

Compound widgets are written in the same way as any other widget application. They are distinguished from regular widget applications in the following ways:

- Compound widgets usually have a base widget at the root of their hierarchies. This base contains the subwidgets that make up the compound widget. From the user's point of view, this single widget *is* the compound widget — its children are not programmatically accessible on their own.

Notice that the base widget at the root of a compound widget is *not* a top-level base. When used, a compound widget must always have a parent widget.

- It is important that the compound widget not make use of the base's user value. In order to preserve the illusion that the compound widget works just like any of the widget primitives, the user value of the compound widget's top-level base should be reserved for use by the caller of the compound widget. Instead, the compound widget should use the user value of one of its child widgets.
- The widget at the root of the compound widget's hierarchy *always* has an event handler function associated with it via the `EVENT_FUNC` keyword to the

widget creating function or the `WIDGET_CONTROL` procedure. This event handler manages events from its sub-widgets and generates events for the compound widget. By swallowing events from the widgets that comprise the compound widget and generating events that represent the compound widget, it presents the illusion that the compound widget is acting like a widget primitive.

- If the compound widget has a value that can be set, it should be assigned a value setting procedure via the `PRO_SET_VALUE` keyword to the widget creating function or the `WIDGET_CONTROL` procedure.
- If the compound widget has a value that can be retrieved, it should be assigned a value retrieving function via the `FUNC_GET_VALUE` keyword to the widget creating function or the `WIDGET_CONTROL` procedure.

For an example of how a compound widget might be written, see [“Example: Compound Widget”](#) on page 49.

The HANDLER Field of the Widget Event Structure

Recall that when `WIDGET_EVENT` finds an event to return, it moves up the widget hierarchy looking for an event-handling routine registered to the widgets in between its current position and the top-level base of the widget application. If such a routine is found, it is called with the event as its argument, and the `HANDLER` field of this event is set to the widget ID of the widget where the event routine was found. Since compound widgets have event handlers associated with their root widget, the `HANDLER` field gives the event handler the widget ID of the root widget. This allows the event handler for a compound widget instance to easily locate the location of its state information relative to this root.

Storing State Information

IDL programmers are often tempted to store the state information directly in the user value of the root widget, but this is not a good idea. The user value of a compound widget is reserved for the user of the widget, just like any basic widget. Therefore, you should store the state information in the user value of one of the child widgets below the root. As a convention, the user value of the first child is often used, leading to event handlers structured as follows:

```
FUNCTION EVENT_FUNC, event
    ; Get state from the first child of the compound widget root:
    child = WIDGET_INFO(event.HANDLER, /CHILD)
    WIDGET_CONTROL, child, GET_UVALUE=state, /NO_COPY

    ; Execute event-handling code here.
```

```
    ; Restore the state information before exiting routine:  
    WIDGET_CONTROL, child, SET_UVALUE=state, /NO_COPY  
  
    ; Return result of function  
    RETURN, result  
END
```

Sometimes, an application will find that it needs to use the user value of all its child widgets for some other purpose, and there is no convenient place to keep the state information. One way to work around this problem is to interpose an extra base between the root base and the rest of the widgets:

```
ROOT = WIDGET_BASE(parent)  
EXTRA = WIDGET_BASE(root)
```

In such an approach, the remaining widgets would all be children of EXTRA rather than ROOT.

Example: Compound Widget

The following example incorporates ideas from the previous sections to show how you might approach the task of writing a compound widget. The widget is called `CW_DICE`, and it simulates a single six-sided die. [Figure 2-1](#) shows the appearance of `XDICE`, an application that uses two instances of `CW_DICE`. `XDICE` is discussed in [“Using `CW_DICE` in a Widget Program”](#) on page 51.

Example Code

The `cw_dice.pro` can be found in the `lib` subdirectory of the IDL distribution. `xdice.pro` can be found in the `examples/doc/widgets` subdirectory of the IDL distribution. Run this example procedure by entering `cw_dice` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT cw_dice.pro`. You should examine these files for additional details and comments not included here. We present sections of the code here for didactic purposes—there is no need to re-create either of these files yourself.

The `CW_DICE` compound widget has the following features:

- It uses a button widget. The current value of the die is displayed as a bitmap label on the button itself. When the user presses the button, the die “rolls” itself by displaying a sequence of bitmaps and then settles on a final value. An event is generated that returns this final value.
- Timer events are used to create the rolling effect. This allows the dice to give the same appearance on machines of varying performance levels. (Timer events are discussed in [“Working with Widget Events”](#) in Chapter 3.)
- The die can be set to a specific value via the `SET_VALUE` keyword to the `WIDGET_CONTROL` procedure. If the desired value is outside of the range 1 through 6, the die is rolled as if the user had pressed the button and a final value is selected randomly. Using `WIDGET_CONTROL` to set the value of the widget in this manner does not cause an event to be issued — IDL’s convention is that user actions cause events, while programmatic changes do not.
- The current value of the die can be obtained via the `GET_VALUE` keyword to the `WIDGET_CONTROL` procedure.

Almost any compound widget will have an associated state. The following information is used by an instantiation of the `CW_DICE` compound widget:

1. The current value.
2. The number of times the die should “tumble” before settling on a final value.

3. The amount of time to take between tumbles.
4. A count of how many tumbles are left before a final value is displayed, while a roll is in progress.
5. The bitmaps to use for the 6 possible die values.
6. The seed to use for the random number generator.

The first four items are stored in a per-widget structure kept in one of the child widget's user values. Since the bitmaps never change, it makes sense to keep them in a COMMON block to be accessed freely by all the CW_DICE routines. It also makes sense to use a single random number seed for the entire CW_DICE class rather than one per instance to avoid the situation where multiple dice, having been created at the same time, have the same seed and thus display the same value on each roll.

Note

It is rare that the use of a COMMON block in a compound widget makes sense. Notice, however, that we are not storing widget state information, but read-only data (bitmaps) and data that can be overwritten at any time with no negative effects (random number generator seed). The use of a COMMON block in this situation means that the read-only data can be created once and used by any number of instantiations of the CW_DICE widget. See [“Managing Application State”](#) on page 42 for a discussion of techniques (including the per-widget structure used here) you can use to store and access widget-specific state information.

Given the above decisions, it is now possible to write the CW_DICE procedure.

Example Code

The following sections discuss elements of the procedure's source code, located in `cw_dice.pro` in the `lib` subdirectory of the IDL distribution. Run this example procedure by entering `cw_dice` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT cw_dice.pro`.

In the CW_DICE function, beginning with `function CW_DICE, parent, value, UVALUE=uvalue`, notice that the code makes reference to two routines named `CW_DICE_SET_VAL` and `CW_DICE_GET_VAL`.

By using the `FUNC_GET_VALUE` and `PRO_SET_VALUE` keywords to `WIDGET_BASE`, `WIDGET_CONTROL` can call these routines whenever the user makes a `WIDGET_CONTROL`, `SET_VALUE` or `GET_VALUE` request. See the functions, `cw_dice_set_val` and `cw_dice_get_val` in the for details.

`CW_DICE_SET_VALUE` makes reference to a procedure named `CW_DICE_ROLL` that does the actual dice rolling. Rolling is implemented as follows:

1. If this is the initial call to `CW_DICE_ROLL`, then pick the final value that will end up being displayed and enter this into the widget's state. Hence, `WIDGET_CONTROL, /GET_VALUE` reports the final value instead of one of the intermediate “tumble” values no matter when it is called.
2. If this is not the final tumble, pick a random intermediate value and display that. Then, make another timer event request for the next tumble.
3. If this is the final tumble, use the saved final value.
4. `CW_DICE_ROLL` works in cooperation with the event handler function for `CW_DICE`. Each timer event causes the event handler to be called and the event handler in turn calls `CW_DICE_ROLL` to process the next tumble.

The `CW_DICE_ROLL` procedure leads us to the event handler function, `CW_DICE_EVENT`. This event handler expects to see button press events generated from a user action as well as `TIMER` events from `CW_DICE_ROLL`. We only want to issue events for the button presses so if the tag name in the event structure is not `WIDGET_TIMER`, then create an event.

Using `CW_DICE` in a Widget Program

We can use `CW_DICE` to implement an application named `XDICE`. `XDICE` displays two dice as well as a “Roll” button. Pressing either die causes it to roll individually. Pressing the “Roll” button causes both dice to roll together. A text widget at the bottom displays the current value.

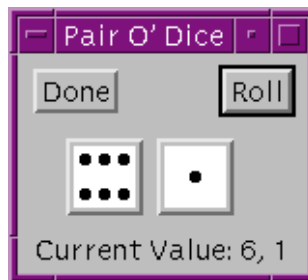


Figure 2-1: The `XDICE` Example Program

Example Code

`xdice.pro` can be found in the `examples/doc/widgets` subdirectory of the IDL distribution. Run this example procedure by entering `xdice` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT xdice.pro`. See [“Running the Example Code”](#) on page 15 if IDL does not run the program as expected.

Debugging Widget Applications

In addition to the “normal” debugging tasks associated with any IDL program, widget applications also require you to debug errors in the widget event loop. If your widget application experiences errors in an event handling routine, keep the following points in mind:

- By default, XMANAGER catches errors and continues processing (see [“CATCH” \(IDL Reference Guide\)](#)). If you are using XMANAGER to manage your widget application (as in most cases you should), calling XMANAGER with CATCH=0 will cause XMANAGER to halt when it encounters an error.

Setting CATCH=0 is useful during debugging, but finished programs should run with the default setting (CATCH=1) and refrain from setting it explicitly.

- CATCH is only effective if XMANAGER is blocking to dispatch errors. During debugging, make sure to call XMANAGER with NO_BLOCK=0 (the default).

Setting NO_BLOCK=0 is useful during debugging, but in many cases you will want your finished program to set NO_BLOCK=1 in order to allow other widget programs (and the IDL command line) to remain active while your application is running.

If a widget application stops responding, you can restart event processing by doing the following:

1. Enter RETALL at the IDL prompt to return to the main program level.
2. Optionally, modify the code to fix the error and re-compile.
3. If one or more of the applications you are running blocks the active command line, enter XMANAGER at the IDL prompt in order to have it resume processing events in the blocking mode. If all applications have NO_BLOCK=1 set, a call to XMANAGER will immediately return, and can be safely omitted.

See [Chapter 8, “Debugging and Error-Handling”](#) for complete details about debugging IDL programs.



Chapter 3

Widget Application Techniques

The following topics are covered in this chapter:

Working with Widget Events	56	Widget Sizing	76
Using Multiple Widget Hierarchies	61	Tips on Creating Widget Applications	82
Creating Menus	64	Enhancing Widget Application Usability ..	84

Working with Widget Events

Widget events and the process of establishing a widget event loop for your application are described in “[Widget Event Processing](#)” on page 34. This section discusses additional topics that may be useful when creating event-driven applications, including:

- “[Interrupting the Event Loop](#)”
- “[Identifying Widget Type from an Event](#)” on page 57
- “[Keyboard Focus Events](#)” on page 57
- “[Timer Events](#)” on page 58
- “[Tracking Events](#)” on page 59
- “[Context Menu Events](#)” on page 60

Interrupting the Event Loop

Beginning with IDL version 5, IDL has the ability to process commands from the IDL command line while simultaneously processing widget events. This means that the IDL command line will remain active even when widget applications are running.

It is possible to interrupt the event function by sending the interrupt character (Control-C). However, you may find that even after sending the interrupt character, IDL does not immediately interrupt the event loop. IDL will interrupt the process that is “on top”—that is, if several applications are running at once, the interrupt will be handled by the first application to receive it.

If your widget application is the only active application, and sending the interrupt does not cause it to break, move the mouse cursor across (or click on) one of the widgets.

This works because when IDL is in the event function, it only checks for the interrupt between event notifications from the window system. Such events do not necessarily translate one-to-one into IDL widget events because the window system typically generates a large number of events related to the window system’s operation that IDL quietly handles. Moving the mouse cursor across the widgets typically generates some of these events which gives IDL a chance to notice the interrupt and act on it.

Identifying Widget Type from an Event

Given a widget event structure, often you need to know what type of widget generated it without having to match the widget ID in the event structure to all the current widgets. This information is available by specifying the `STRUCTURE_NAME` keyword to the `TAG_NAMES` function:

```
PRINT, 'Event structure type: ', TAG_NAMES(EVENT, /STRUCTURE_NAME)
```

This works because each widget type generates a different event structure. The event structure generated by a given widget type is documented in the description of the widget creation function in the *IDL Reference Guide*.

When using this technique, be aware that although all the basic widgets use named structures for their events, many compound widgets return anonymous structures. This technique does not work well in that case because anonymous structures lack a recognizable name.

An alternative technique involves using the `TYPE` keyword to the [WIDGET_INFO](#) function. This method is useful when the widget event name does not specify the widget from which the event originated. Timer events are an example; although the events originate from a widget, the event structure's name is `WIDGET_TIMER`. The following statement checks to see if the event is a timer event and, if it is, prints the type code of the widget that generated the event.

```
IF ((TAG_NAMES(EVENT, /STRUCTURE) EQ 'WIDGET_TIMER') THEN $
PRINT, WIDGET_INFO(EVENT.ID, /TYPE)
```

Such a check would be useful if a given widget could generate *either* a timer event or a “normal” event, and you wanted to differentiate between the two.

Note

Always check for a distinct type of widget event. IDL will continue to add new widgets with new event structures, so it is important not to make assumptions about the contents of a random widget event structure. The structure of existing widget events will remain stable, (although new fields may be added) so checking for a particular type of widget event will always work.

Keyboard Focus Events

Base, table, and text widgets can be set to generate *keyboard focus events*. Generating and examining keyboard focus events allows you to determine when a given widget has either *gained* or *lost* the keyboard focus—that is, when it is brought to the foreground or when it is covered by another window.

Set the `KBRD_FOCUS_EVENTS` keyword to `WIDGET_BASE`, `WIDGET_TABLE`, or `WIDGET_TEXT` to generate keyboard focus events. (You can also modify an existing base, table, or text widget to generate keyboard focus events using the `KBRD_FOCUS_EVENTS` keyword to `WIDGET_CONTROL`.) You can then use your event-handling procedure to cache the widget ID of the last widget (with keyboard focus events enabled) to have the keyboard focus. One situation where this is useful is when you have an application menu (created with the `MBAR` keyword to `WIDGET_BASE`) and you wish to perform an action in a text widget based on the menu item selected. Although the event generated by the user's menu selection has the *menu's* base as its top-level widget ID, if you generate and track keyboard focus events for the text widget, you can “remember” which widget the action triggered by the menu selection should affect. Note that in this example, keyboard focus events are *not* generated for the menubar's base.

Timer Events

In addition to the normal widget events discussed previously, IDL allows the user to make *timer event* requests by using the `TIMER` keyword. Such events are useful in many applications that are time dependent, such as animation. The syntax for making such a request is:

```
WIDGET_CONTROL, Widget_Id, TIMER=interval_in_seconds
```

Widget_Id can be the ID of any type of widget. When such a request is made, IDL generates a timer request after the requested time interval has passed. Timer events consist of a structure with only the standard three fields — no additional information is provided.

Note

At most one timer event request can be associated with a given widget ID. If multiple timer event requests are associated with a single widget, the last request made takes precedence.

It is up to the programmer to differentiate between a normal event and a timer event for a given widget. One way to solve this problem is to make timer requests for widgets that do not otherwise generate events, such as base or label widgets.

Each timer request causes a single event to be generated. To generate a steady stream of timer events, you must make a new timer request in the event handler routine each

time a timer event is delivered. The following example demonstrates how to check for a timer event and generate a new timer event each time a timer event occurs:

```

PRO timer_example_event, ev

    WIDGET_CONTROL, ev.ID, GET_UVALUE=uval
    IF (TAG_NAMES(ev, /STRUCTURE_NAME) EQ 'WIDGET_TIMER') THEN BEGIN
        PRINT, 'Timer Fired'
        WIDGET_CONTROL, ev.TOP, TIMER=2
    ENDIF

    CASE uval OF
        'timer' : BEGIN
            WIDGET_CONTROL, ev.TOP, TIMER=2
            END
        'exit' : WIDGET_CONTROL, ev.TOP, /DESTROY
    ELSE:
    ENDCASE

END

PRO timer_example
    base = WIDGET_BASE(/COLUMN, UVALUE='base')
    b1 = WIDGET_BUTTON(base, VALUE='Fire event', UVALUE='timer')
    b2 = WIDGET_BUTTON(base, VALUE='Exit', UVALUE='exit')
    WIDGET_CONTROL, base, /REALIZE
    XMANAGER, 'timer_example', base, /NO_BLOCK
END

```

See [“Draw Widget Example”](#) on page 118 for a larger example using timer events.

Tracking Events

Tracking events allow you to determine when the mouse pointer has entered or left the area of the computer screen covered by a given widget. You can use tracking events to allow your interface to react as the user moves the mouse pointer over different interface elements. Tracking events are generated for a widget when the widget creation routine is called with the `TRACKING_EVENTS` keyword set.

The event structure of a tracking event includes a field named `ENTER` that contains a 1 (one) if the mouse pointer entered the region covered by the widget, or 0 (zero) if the mouse pointer left the region covered by the widget. The following example demonstrates how to check for tracking events and modify the value of a button widget when the mouse cursor is positioned over it.

```

PRO tracking_demo_event, event
    IF (TAG_NAMES(event, /STRUCTURE_NAME) EQ 'WIDGET_TRACKING') $
    THEN BEGIN

```

```

        IF (event.ENTER EQ 1) THEN BEGIN
            WIDGET_CONTROL, event.ID, SET_VALUE='Press to Quit'
        ENDIF ELSE BEGIN
            WIDGET_CONTROL, event.ID, $
                SET_VALUE='What does this button do?'
            ENDELSE
        ENDIF ELSE BEGIN
            WIDGET_CONTROL, event.TOP, /DESTROY
        ENDELSE
    END

    PRO tracking_demo
        base = WIDGET_BASE(/COLUMN)
        button = WIDGET_BUTTON(base, $
            VALUE='What does this button do?', /TRACKING_EVENTS)
        WIDGET_CONTROL, base, /REALIZE
        XMANAGER, 'tracking_demo', base
    END

```

Context Menu Events

Base, list, text, table and tree widgets can be set to generate *context menu events*. Generating and examining context menu events allows you to determine when the user has clicked the right-hand mouse button over a given widget, which in turn allows you to display a “context menu.” (Draw widgets can also generate events when the right-hand mouse button is clicked, using the general `BUTTON_EVENTS` mechanism.) See “[Context-Sensitive Menus](#)” on page 69 for a detailed description.

Using Multiple Widget Hierarchies

Using widgets, you can create IDL applications with graphical user interfaces. Although widget applications are running “inside” IDL, a well-designed program can behave and appear just like a stand-alone application.

While a simple application may consist of a single widget hierarchy headed by a single top-level base widget, more complex applications can include multiple widget hierarchies, each with their own top-level base. Widget applications that include multiple widget hierarchies consist of a *group* of top-level base widgets organized hierarchically. The individual widgets that make up the widget application’s interface have as their parent widget either one of the top-level bases or a base that is a child of one of the top-level bases.

Groups of widgets are defined by setting the `GROUP_LEADER` keyword when creating the widget. Group membership controls how and when widgets are iconized, which layer they appear in, and when they are destroyed.

Figure 3-1 depicts a widget application group hierarchy consisting of six top-level bases in three groups: base 1 leads all six bases, base 2 leads bases 4 and 5, and base 3 leads base 6. What does this mean? Operations like iconization or destruction that affect base 2 also affect bases 4 and 5. Operations that affect base 3 also affect base 6. Operations that affect base 1 affect all six bases—that is, a group includes not only those bases that explicitly claim one base as their leader, but also all bases *led by* those member bases.

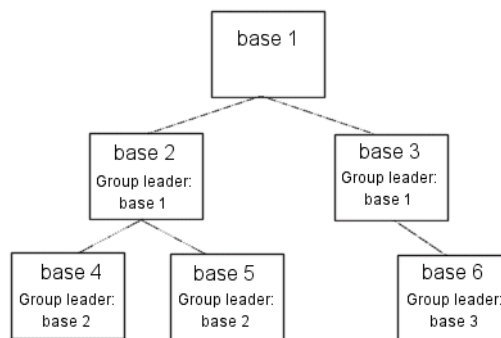


Figure 3-1: A widget application group hierarchy with six top-level bases.

The following IDL commands would create this hierarchy:

```
base1 = WIDGET_BASE ()
base2 = WIDGET_BASE (GROUP_LEADER=base1)
base3 = WIDGET_BASE (GROUP_LEADER=base1)
base4 = WIDGET_BASE (GROUP_LEADER=base2)
base5 = WIDGET_BASE (GROUP_LEADER=base2)
base6 = WIDGET_BASE (GROUP_LEADER=base3)
```

Widget Group Behaviors

Groups of widgets are displayed and destroyed according to the following principles:

Iconization

Bases and groups of bases can be *iconized* (or *minimized*) by clicking the system minimize control. When a group leader is iconized, all members of the group are minimized as well.

Layering

Layering is the process by which groups of widgets seem to share the same plane on the display screen. Within a layer on the screen, widgets have a *Z-order*, or front-to-back order, that defines which widgets appear to be on top of other widgets.

All widgets within a group hierarchy share the same layer—that is, when one group member has the input focus, all members of the group hierarchy are displayed in a layer that appears in front of all other groups or applications. Within the layer, the widgets can have an arbitrary Z-order, determined by the programmer.

Destruction

When a group leader widget is destroyed, either programmatically or by clicking on the system “close” button, all members of the group are destroyed as well.

See [“Iconizing, Layering, and Destroying Groups of Top-Level Bases”](#) under [“WIDGET_BASE”](#) (*IDL Reference Guide*) for detailed information on how group membership defines widget behavior on different platforms.

Titlebar Icon Inheritance

On Microsoft Windows platforms, if the group leader has the `BITMAP` keyword set, the same titlebar icon is used for all group members. (Titlebar icons are not supported on Motif platforms.)

Floating bases

Top-level base widgets created with the FLOATING keyword set will *float* above their group leaders, even though they share the same layer. Floating bases and their group leaders are iconized in a single icon (on platforms where iconization is possible). Floating bases are destroyed when their group leaders are destroyed.

Modal bases

Top-level base widgets created with the MODAL keyword will float above their group leaders, and will suspend processing in the widget application until they are dismissed. (*Dialogs* are generally modal.) Modal bases cannot be iconized, and on some platforms other bases cannot be moved or iconized while the modal dialog is present. Modal bases cannot have scroll bars or menubars.

Menubars

Widget applications can have an application-specific menubar, created by the MBAR keyword to WIDGET_BASE. Menus and menubars are discussed in detail in [“Creating Menus”](#) on page 64.

Creating Menus

Menus allow a user to select one or more options from a list of options. IDL widgets allow you to build a number of different types of menus for your widget application.

This section discusses the following different types of menus:

- [Button Groups](#)
- [Lists](#)
- [Pulldown Menus](#)
- [Menus on Top-Level Bases](#)
- [Context-Sensitive Menus](#)

Button Groups

One approach to menu creation is to build an array of buttons. With a button menu, all options are visible to the user all the time. To create a button menu, do the following:

1. Call the `WIDGET_BASE` function to create a base to hold the buttons. Use the `COLUMN` and `ROW` keywords to determine the layout of the buttons.
2. Call the `WIDGET_BUTTON` function once for each button to be added to the base created in the previous step.

Because menus of buttons are common, IDL provides a compound widget named `CW_BGROU`P to create them. Using `CW_BGROU`P rather than a series of calls to `WIDGET_BUTTON` simplifies creation of a menu of buttons and also simplifies event handling by providing a single event structure for the group of buttons. For example, the following IDL statements create a button menu with five choices:

```
values = ['One', 'Two', 'Three', 'Four', 'Five']
base = WIDGET_BASE()
bgroup = CW_BGROU(base, values, /COLUMN)
WIDGET_CONTROL, base, /REALIZE
```

In this example, one call to `CW_BGROU`P replaces five calls to `WIDGET_BUTTON`.

Exclusive or Nonexclusive Buttons

Buttons in button groups normally act as independent entities, returning a selection event (a one in the select field of the event structure) or similar value when pressed. Groups of buttons can also be made to act in concert, as either exclusive or non-

exclusive groups. In contrast to normal button groups, both exclusive and non-exclusive groups display which buttons have been selected.

Exclusive button groups allow only one button to be selected at a given time. Clicking on an unselected button deselects any previously-selected buttons. *Non-exclusive* button groups allow any number of buttons to be selected at the same time. Clicking on the same button repeatedly selects and deselects that button. The following code creates three button groups. The first group is a “normal” button group as created in the previous example. The next is an exclusive group, and the third is a non-exclusive group.

```
values = ['One', 'Two', 'Three', 'Four', 'Five']
base = WIDGET_BASE(/ROW)
bgroup1 = CW_BGROUP(base, values, /COLUMN, $
    LABEL_TOP='Normal', /FRAME)
bgroup2 = CW_BGROUP(base, values, /COLUMN, /EXCLUSIVE, $
    LABEL_TOP='Exclusive', /FRAME)
bgroup3 = CW_BGROUP(base, values, /COLUMN, /NONEXCLUSIVE, $
    LABEL_TOP='Nonexclusive', /FRAME)
WIDGET_CONTROL, base, /REALIZE
```

The widget created by this code is shown in the following figure:

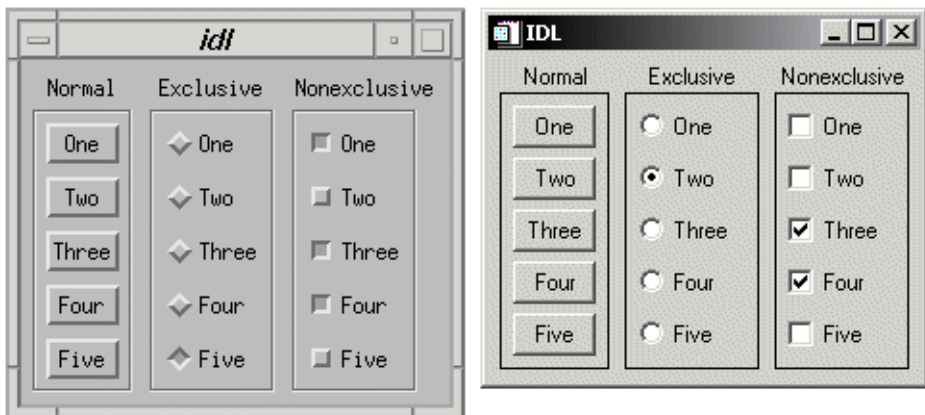


Figure 3-2: Normal Menus (left), Exclusive Menus (center) and Non-exclusive Menus (right)

Lists

A second approach to menu creation is to provide the user with a list of options in the form of a scrolling or drop-down list. A scrolling list is always displayed, although it may not show all items in the list at all times. A drop-down list shows only the selected item until the user clicks on the list, at which time it displays the entire list. Both lists allow only a single selection at a time.

The following example code uses the `WIDGET_LIST` and `WIDGET_DROPLIST` functions to create two menus of five items each. While both lists contain five items, the scrolling list displays only three at a time, because we specify this with the `YSIZE` keyword.

```
values = ['One', 'Two', 'Three', 'Four', 'Five']
base = WIDGET_BASE(/ROW)
list = WIDGET_LIST(base, VALUE=values, YSIZE=3)
drop = WIDGET_DROPLIST(base, VALUE=values)
WIDGET_CONTROL, base, /REALIZE
```

The widget created by this code is shown in the following figure:

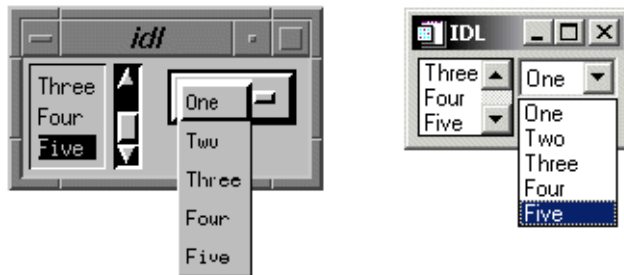


Figure 3-3: Scrolling and drop-down lists.

Pulldown Menus

A third approach to menu creation involves menus that appear as a single button until the user selects the menu, at which time the menu pops up to display the list of possible selections. Buttons in such a pulldown menu can activate other pulldown menus to any desired depth. The method for creating a pulldown menu is as follows:

1. The topmost element of any pulldown menu is a button, created with the `MENU` keyword to the `WIDGET_BUTTON` function.

2. The top-level button has one or more child widget buttons attached. (That is, one or more buttons specify the first button's widget ID as their "parent.") Each button can either be used as is, in which case pressing it causes an event to be generated, or it can be created with the MENU keyword and have further child widget buttons attached to it. If it has child widgets, pushing it causes a pulldown menu containing the child buttons to pop into view.
3. Menu buttons can be the parent of other buttons to any desired depth.

Because pulldown menus are common, IDL provides a compound widget named `CW_PDMENU` to create them. Using `CW_PDMENU` rather than a series of calls to `WIDGET_BUTTON` simplifies creation of a pulldown menu in the same way the `CW_BGROU`P simplifies the creation of button menus.

The following example uses `CW_PDMENU` to create a pulldown menu. First, we create an array of anonymous structures to contain the menu descriptions.

```
desc = REPLICATE({ flags:0, name:'' }, 6)
```

The `desc` array contains six copies of the empty structure. Each structure has two fields: `flags` and `name`. Next, we populate these fields with values:

```
desc.flags = [ 1, 0, 1, 0, 2, 2 ]
desc.name = [ 'Operations', 'Predefined', 'Interpolate', $
             'Linear', 'Spline', 'Quit' ]
```

The value of the `flags` field specifies the role of each button. In this example, the first and third buttons start a new sub-menu (values are 1), the second and fourth buttons are plain buttons with no other role (values are 0), and the last two buttons end the current sub-menu and return to the previous level (values are 2). The value of the `name` field is the value (or label) of the button at each level.

```
base = WIDGET_BASE()
menu = CW_PDMENU(base, desc)
WIDGET_CONTROL, base, /REALIZE
```

The format of the menu description used by `CW_PDMENU` in the above example requires some explanation. `CW_PDMENU` views a menu as consisting of a series of buttons, each of which can optionally lead to a sub-menu. The description of each button consists of a structure supplying its name and a flag field that tells what kind of button it is (starts a new sub-menu, ends the current sub-menu, or a plain button within the current sub-menu). The description of the complete menu consists of an array of such structures corresponding to the flattened menu.

Compare the description used in the code above with the result shown in the following figure.

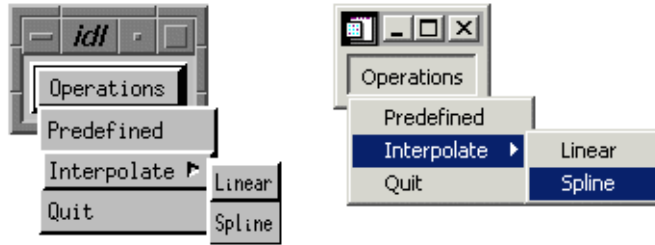


Figure 3-4: Pulldown menus created with `CW_PDMENU`.

Menus on Top-Level Bases

A fourth approach to providing menus in your widget application is to attach the menus directly to the top-level base widget. Menus attached to a top-level base widget are created just like pulldown menus created from button widgets, but they do not appear as buttons. Menus created in this way are children of a special sub-base of the top-level base, created by specifying the `MBAR` keyword when the top-level base is created.

For example, the following code creates a top-level base widget and attaches a menu titled `MENU1` to it. `MENU1` contains the choices `ONE`, `TWO`, and `THREE`.

```
base = WIDGET_BASE(MBAR=bar)
menu1 = WIDGET_BUTTON(bar, VALUE='MENU1', /MENU)
button1 = WIDGET_BUTTON(menu1, VALUE='ONE')
button2 = WIDGET_BUTTON(menu1, VALUE='TWO')
button3 = WIDGET_BUTTON(menu1, VALUE='THREE')
draw = WIDGET_DRAW(base, XSIZE=100, YSIZE=100)
WIDGET_CONTROL, base, /REALIZE
```

The resulting widget is shown in the following figure:



Figure 3-5: Menus attached to a top-level base.

Context-Sensitive Menu

Context-sensitive menus (also referred to as *context menu* or *pop-up menu*) are hidden until a user performs an action to display the menu. When summoned, the appearance of a context menu is similar to that of a menu created in a floating, modal base. The behavior of a context menu is the same as that of a menu on a menu bar; when the user clicks one of the menu's buttons, a button event is generated and the menu is dismissed. If the user clicks outside the context menu, it is dismissed without generating any events.

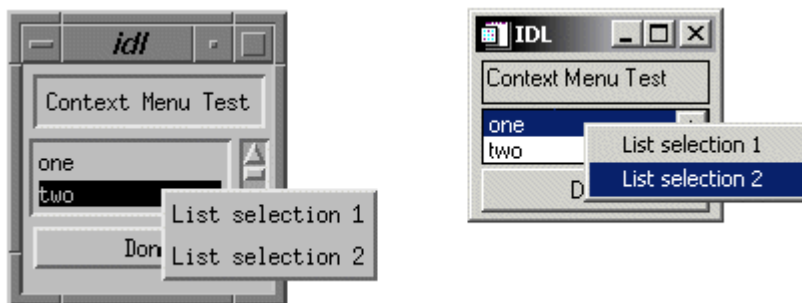


Figure 3-6: Widget Context Menus.

By convention, context-sensitive menus in IDL widget applications are displayed when the user clicks the right mouse button. Several IDL widget primitives — base, draw, list, table, text, and tree widgets — can be configured to generate events when this occurs. The mechanism used to generate right mouse button events is different for draw widgets than for the other types; these differences are discussed below.

Note

While it is customary to display context-sensitive menus when the user clicks the right mouse button, IDL's mechanism for displaying the menus is quite general, and can be invoked under many circumstances. Examples in this section will discuss the common usage.

To create a context-sensitive menu in a widget application, do the following:

1. Create a Context Menu
2. Generate and Handle Context Events
3. Display the Context Menu
4. Process Button Events

Create a Context Menu

Context menus are contained within a special base widget created with the `CONTEXT_MENU` keyword. A base widget used as the base for a context menu must have as its parent widget one of the following widget types:

- Base widget
- Draw widget
- List widget
- Property sheet widget
- Table widget
- Text widget
- Tree widget

The process for creating a context menu is similar to that for creating a menu for a top-level base (a menubar). Create menu entries on the base widget using the `WIDGET_BUTTON` function. Context menu entries can display sub-menus (using the `MENU` keyword to `WIDGET_BUTTON` or the `CW_PDMENU` compound widget) or appear as separators (using the `SEPARATOR` keyword to `WIDGET_BUTTON`).

The following code snippet illustrates a very simple context menu attached to a base widget:

```
topLevelBase = WIDGET_BASE(/CONTEXT_EVENTS)
contextBase = WIDGET_BASE(topLevelBase, /CONTEXT_MENU)
button1 = WIDGET_BUTTON(contextBase, VALUE='First button')
button2 = WIDGET_BUTTON(contextBase, VALUE='Second button')
```

Generate and Handle Context Events

Generating Right Mouse Button Events

In order to display the context menu at the appropriate time, the widget that serves as the parent for the context menu base must be configured to generate an event when the user clicks the right mouse button over that widget. For base, list, property sheet, table, text, and tree widgets, this is accomplished by setting the `CONTEXT_EVENTS` keyword when creating the widget, or by enabling context events by setting the `CONTEXT_EVENTS` keyword to `WIDGET_CONTROL`. When a user clicks the right mouse button over an appropriately configured base, list, text, or tree widget, a widget event with the following structure is generated:

```
{WIDGET_CONTEXT, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L,
  ROW:0L, COL:0L}
```

The first three fields are the standard fields found in every widget event. The `X` and `Y` fields give the device coordinates at which the event occurred, measured from the upper left corner of the base widget. The `ROW` and `COL` fields return meaningful information for table widgets and values of zero (0) for other widgets.

For table widgets, `ROW` and `COL` indicate the zero-based index of the cell that was clicked on when the context menu was initiated. The upper-left data cell has a row and column index of 0,0. Row and column headers have indices of -1. If the context menu event takes place outside of all table cells and headers, then both `ROW` and `COL` will have values of -1.

Note

When working with context menu events, it is important to notice that the event structure does not have a `TYPE` field, so special code is needed for the property sheet event handler. Instead of keying off the `TYPE` field, use the structure's name. An example is provided in the `WIDGET_PROPERTY SHEET` "Example" section in the *IDL Reference Guide*.

For draw widgets, button events are handled differently. Set the `BUTTON_EVENTS` keyword to `WIDGET_DRAW` (or the `DRAW_BUTTON_EVENTS` keyword to `WIDGET_CONTROL`) to generate widget events with the following structure:

```
{ WIDGET_DRAW, ID:0L, TOP:0L, HANDLER:0L, TYPE: 0, X:0L, Y:0L,
  PRESS:0B, RELEASE:0B, CLICKS:0, MODIFIERS:0L, CH:0, KEY:0L }
```

The first three fields are the standard fields found in every widget event. The `X` and `Y` fields give the device coordinates at which the event occurred, measured from the lower left corner of the drawing area. `PRESS` and `RELEASE` are bitmasks that represent which of the left, center, or right mouse button was pressed: that is, a value of 1 (one) represents the left button, 2 represents the middle button, and 4 represents the right button. (See [“Widget Events Returned by Draw Widgets”](#) (*IDL Reference Guide*) for a complete description of the `WIDGET_DRAW` event structure.)

Detecting Right Mouse Button Events

Once the parent widget of your context menu is configured to generate events when the user clicks the right mouse button, you must detect the events in your event handler routine. For base, list, text, and tree widgets, your event handler should examine the event structure name to determine the type of event; if the event is of type `WIDGET_CONTEXT`, you know that the right mouse button was pressed.

To detect a right mouse button click in a base, list, text, or tree widget (with context events enabled), use the following test:

```
IF (TAG_NAMES(event, /STRUCTURE_NAME) EQ 'WIDGET_CONTEXT') THEN
BEGIN
    ; process event here
ENDIF
```

For draw widgets, your event handler should examine the `WIDGET_DRAW` event structure; if the value of the `RELEASE` field is equal to four, you know that the right mouse button was pressed and released.

To detect a right mouse button click in a draw widget (with button events enabled), use the following test:

```
IF (event.release EQ 4) THEN BEGIN
    ; process event here
ENDIF
```


Note that in a complex widget application, your event handler may first need to determine whether the event came from a draw widget. In this case, you may need a test that looks like this:

```
IF (TAG_NAMES(event, /STRUCTURE_NAME) EQ 'WIDGET_DRAW') THEN BEGIN
    IF (event.release EQ 4) THEN BEGIN
        ; process event here
    ENDIF
ENDIF
```

Display the Context Menu

When your event handler routine detects a right mouse button click, use the `WIDGET_DISPLAYCONTEXTMENU` procedure to display the context menu. This routine takes as its parameters the widget ID of the widget for which the context menu is to be displayed, the X and Y coordinates at which the menu should be displayed, and the widget ID of the context menu base widget that holds the context menu. See [“WIDGET_DISPLAYCONTEXTMENU”](#) (*IDL Reference Guide*) for additional information.

In all cases, the ID field of the event structure generated by the right mouse button click contains the widget ID of the widget whose context menu is to be displayed. Similarly, the event structure contains the location of the mouse click in the X and Y fields; in most cases, this is where you will want to display the context menu.

The following code fragment would display a context menu held in a base widget whose widget ID is `contextBase` at the location of the user’s right mouse click:

```
WIDGET_DISPLAYCONTEXTMENU, event.ID, event.X, $
    event.Y, contextBase
```

In a simple application with only one context menu, you know the widget ID of the context menu base widget to be displayed. In a real application, however, it is likely that more than one context menu exists. See [“Determining Which Context Menu to Display”](#), below, for tips on dealing with multiple context menus.

Process Button Events

Once the context menu is displayed, processing events that flow from it is the same as processing events from any other menu. The individual buttons that make up the menu can have event handler routines associated with them; these routines are then invoked when the user clicks on one of the menu buttons. See the [“Context Menu Example”](#) below for a simple illustration of menu button event processing.

Determining Which Context Menu to Display

In a real application, you may have multiple context menus available to display when the user right-clicks on different portions of the user interface. One way to handle this situation is to have your event handler keep track of which context menu should be displayed for each widget. Consider a widget hierarchy that contains a text widget and a list widget, both of which have associated context menus:

```

topLevelBase = WIDGET_BASE(/COLUMN, XSIZE = 120, YSIZE = 80)
wText = WIDGET_TEXT(topLevelBase, VALUE="Context Menu Test", $
    /CONTEXT_EVENTS)
wList = WIDGET_LIST(topLevelBase, VALUE=['one', 'two', 'three'], $
    /CONTEXT_EVENTS)
contextBase1 = WIDGET_BASE(wText, /CONTEXT_MENU, $
    UNAME="tContextMenu")
contextBase2 = WIDGET_BASE(wList, /CONTEXT_MENU, $
    UNAME="lContextMenu")

```

Now the application's event handler, after detecting a right mouse button click with the

```

IF (TAG_NAMES(event, /STRUCTURE_NAME) EQ 'WIDGET_CONTEXT')

```

test, must somehow determine whether the user had clicked on the text widget or the list widget. To make this determination, you could use the `WIDGET_INFO` function to search the widget hierarchy starting with the widget at the top of the event structure for a widget with the correct `UNAME` value:

```

IF (WIDGET_INFO(event.id, FIND_BY_UNAME = 'tContextMenu') GT 0) $
    THEN BEGIN
        WIDGET_DISPLAYCONTEXTMENU, event.id, event.x, event.y, $
            WIDGET_INFO(event.id, FIND_BY_UNAME = 'tContextMenu')
    ENDIF
IF (WIDGET_INFO(event.id, FIND_BY_UNAME = 'lContextMenu') GT 0) $
    THEN BEGIN
        WIDGET_DISPLAYCONTEXTMENU, event.id, event.x, event.y, $
            WIDGET_INFO(event.id, FIND_BY_UNAME = 'lContextMenu')
    ENDIF

```

While this method will always work, it may involve a substantial amount of code, and must search the widget hierarchy multiple times to find the widget ID of the base for the context menu. If, however, your application has at most one context menu for each base, draw, list, or text widget, you can streamline the code significantly by using a common `UNAME` value for all of the context menus. For example, if the definitions of the context menu bases change to this:

```

contextBase1 = WIDGET_BASE(wText, /CONTEXT_MENU, $
    UNAME="contextMenu")
contextBase2 = WIDGET_BASE(wList, /CONTEXT_MENU, $

```

```
UNAME="contextMenu")
```

then the code detecting and displaying the context menu becomes:

```
contextBase = WIDGET_INFO(event.ID, FIND_BY_UNAME = 'contextMenu')

WIDGET_DISPLAYCONTEXTMENU, event.ID, event.X, $
event.Y, contextBase
```

Since the context menu base is a child of the text or list widget, the call to `WIDGET_INFO` finds the appropriate base by searching for the `UNAME` value “contextMenu”, starting at the widget specified by `event.ID`.

Context Menu Example

The context menu example defines a simple application with two context menus, one each for a list widget and a text widget. When a menu item on one of the context menus is selected, IDL prints an informational message.

Example Code

See the file `context_menu_example.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution for the example code. Run this example procedure by entering `context_menu_example` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT context_menu_example.pro`. See [“Running the Example Code”](#) on page 15 if IDL does not run the program as expected.

Example Code

Additional examples using the context menu in various situations can be found in the `examples/doc/widgets` subdirectory of the IDL distribution:

```
context_tlbases_example.pro
context_draw_example.pro
context_list_example.pro
context_text_example.pro
```

Widget Sizing

This section explains how IDL widgets size themselves, widget geometry concepts, and how to explicitly size and position widgets.

Widget Geometry Terms and Concepts

Widget size and layout is determined by many interrelated factors. In the following discussion, the following terms are used:

- *Geometry*: The size and position of a widget.
- *Natural Size*: The natural, or implicit, size of a widget is the size a widget has if no external constraints are placed on it. For example, a label widget has a natural size that is determined by the size of the text it is displaying and space for margins. These values are influenced by such things as the size of the font being displayed and characteristics of the low-level (i.e., operating-system level) widget or control used to implement the IDL widget.
- *Explicit Size*: The explicit, or user-specified, size of a widget is the size set when an IDL programmer specifies one of the size keywords to an IDL widget creation function or `WIDGET_CONTROL`.

Note

To retrieve information about the geometry of an existing widget, use the `GEOMETRY` keyword to the `WIDGET_INFO` function.

How Widget Geometry is Determined

IDL uses the following rules to determine the geometry of a widget:

- The explicit size of a widget, if one is specified, takes precedence over the natural size. That is, the user-specified size is used if available.
- If an explicit size is not specified, the natural size of the widget—at the time the widget is realized—is used. Once realized, the size of a widget *does not automatically change* when the value of the widget changes, unless the widget's dynamic resize property has been set. Dynamic resizing is discussed in more detail below. Note that any realized widget can be made to change its size by calling `WIDGET_CONTROL` with any of the sizing keywords.

- Children of a “bulletin board” base (i.e., a base that was created without setting the COLUMN or ROW keywords) have an offset of (0,0) unless an offset is explicitly specified via the XOFFSET or YOFFSET keywords.
- The offset keywords to widgets that are children of ROW or COLUMN bases are ignored, and IDL calculates the offsets to lay the children out in a grid. This calculation can be influenced by setting any of the ALIGN or BASE_ALIGN keywords when the widgets are created.

Dynamic Resizing

Realized widgets, by default, do not automatically resize themselves when their values change. This is true whether the widget was created with an explicit size or the widget was allowed to size itself naturally. This behavior makes it easy to create widget layouts that don’t change size too frequently or “flicker” due to small changes in a widget’s natural size.

This default behavior can be changed for label, button, and droplist widgets. Set the DYNAMIC_RESIZE keyword to WIDGET_LABEL, WIDGET_BUTTON, or WIDGET_DROPLIST to make a widget that automatically resizes itself when its value changes. Note that the XSIZE and YSIZE keywords should not be used with DYNAMIC_RESIZE. Setting explicit sizing values overrides the dynamic resize property and creates a widget that *will not* resize itself.

Explicitly Specifying the Size and Location of Widgets

The XSIZE (and SCR_XSIZE), YSIZE (and SCR_YSIZE), XOFFSET, and YOFFSET keywords, when used with a standard base widget parent (a base created without the COLUMN or ROW keywords—also called a “bulletin board” base), allow you to specify exactly how the child widgets should be positioned. Sometimes this is a very useful option. However, in general, it is best to avoid this style of programming. Although these keywords are usually honored, they are merely hints to the widget toolkit and might be ignored.

Note

Draw widgets are the exception to this recommendation. In almost all cases, you will want to set the size of draw widgets explicitly, using the sizing keywords.

Explicitly specifying the size and offset makes a program inflexible and unable to run gracefully on various platforms. Often, a layout of this type will look good on one platform, but variations in screen size and how the toolkit works will cause widgets to

overlap and not look good on another platform. The best way to handle this situation is to use nested row and column bases to hold the widgets and let the widgets arrange themselves. Such bases are created using the COLUMN and ROW keywords to the WIDGET_BASE function.

Sizing Keywords

When explicitly setting the size of a widget, IDL allows you to control three aspects of the size:

- The *virtual size* is the size of the *potentially* viewable area of the widget. The virtual size may be larger than the actual viewable area on your screen. The virtual size of a widget is determined by either the widget's value, or the XSIZE and YSIZE keywords to the widget creation routine.
- The *viewport size* is the size of the viewable area on your screen. If the viewport size is smaller than the virtual size, scroll bars may be present to allow you to view different sections of the viewable area. When creating widgets for which scroll bars are appropriate, you can add scroll bars by setting the either SCROLL keyword or the APP_SCROLL keyword to the widget creation routine. (For information on the difference, see [“Scrolling Draw Widgets”](#) on page 113.) You can explicitly set the size of the viewport area using the X_SCROLL_SIZE and Y_SCROLL_SIZE keywords when creating base, draw, and table widgets.
- The *screen size* is the size of the widget on your screen. You can explicitly specify a screen size using the SCR_XSIZE and SCR_YSIZE keywords to the widget creation routine. Explicitly-set viewport sizes (set with X_SCROLL_SIZE or Y_SCROLL_SIZE) are ignored if you specify the screen size.

The following code shows an example of the WIDGET_DRAW command:

```
draw = WIDGET_DRAW(base, XSIZE=384, YSIZE=384,$
  X_SCROLL_SIZE=192, Y_SCROLL_SIZE = 192, SCR_XSIZE=200)
```

This results in the following:

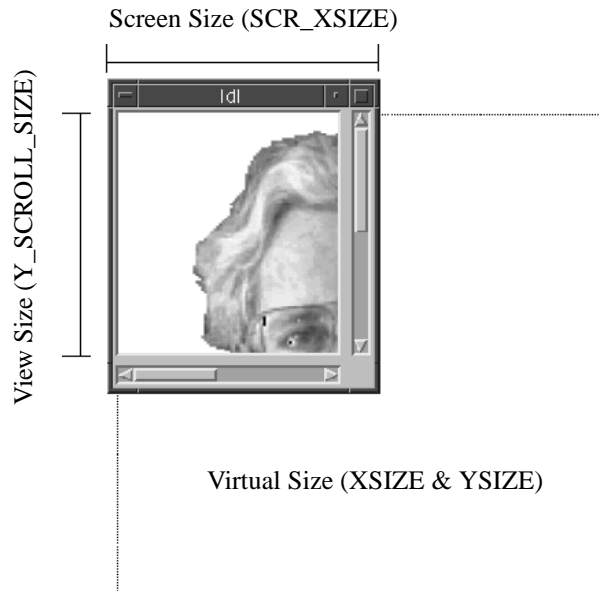


Figure 3-7: Visual description of widget sizes.

In this case, the `XSIZE` and `YSIZE` keywords set the virtual size to 384 x 384 pixels. The `X_SCROLL_SIZE` and `Y_SCROLL_SIZE` keywords set the viewport size to 192 x 192 pixels. Finally, the `SCR_XSIZE` keyword overrides the `X_SCROLL_SIZE` keyword and forces the screen size of the widget (in the X-dimension) to 200 pixels, including the scroll bar.

Controlling Widget Size after Creation

A number of keywords to the `WIDGET_CONTROL` procedure allow you to change the size of a widget after it has been created. (You will find a list of the keywords to `WIDGET_CONTROL` that apply to each type of widget at the end of the widget creation routine documentation.) Note that keywords to `WIDGET_CONTROL` may not control the same parameters as their counterparts associated with widget creation routines. For example, while the `XSIZE` and `YSIZE` keywords to `WIDGET_DRAW` control the virtual size of the draw widget, the `XSIZE` and `YSIZE` keywords to `WIDGET_CONTROL` (when called with the widget ID of a draw widget) control the viewport size of the draw widget. See the *IDL Reference Guide* for details.

Units of Measurement

You can specify the unit of measurement used for most widget sizing operations. When using a widget creation routine, or when using `WIDGET_CONTROL` or `WIDGET_INFO`, set the `UNITS` keyword equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

Note

The `UNITS` keyword does not affect all sizing operations. Specifically, the value of `UNITS` is ignored when setting the `XSIZE` or `YSIZE` keywords to `WIDGET_LIST`, `WIDGET_TABLE`, or `WIDGET_TEXT`.

Finding the Size of the Screen

When creating the top-level base for an application, sometimes it is useful to know the size of the screen. This information is available via the `GET_SCREEN_SIZE` function. `GET_SCREEN_SIZE` returns a two-element floating-point array specifying the size of the screen, in pixels. See “`GET_SCREEN_SIZE`” (*IDL Reference Guide*) for details.

Preventing Layout Flicker

After a widget hierarchy has been realized, adding or destroying widgets in that hierarchy causes IDL to recalculate and set new geometries for every widget in the hierarchy. When a number of widgets are added or destroyed, these calculations occur between each change to the hierarchy, resulting in unpleasant screen “flashing” as the user sees a brief display of each intermediate widget configuration. This behavior can be eliminated by using the `UPDATE` keyword to `WIDGET_CONTROL`.

The top-level base of every widget hierarchy has an `UPDATE` attribute that determines whether or not changes to the hierarchy are displayed on screen. Setting `UPDATE` to 0 turns off immediate updates and allows you to make a large number of changes to a widget hierarchy without updating the screen after each change. After all of your changes have been made, setting `UPDATE` to 1 causes the final widget configuration to be displayed on screen.

For example, consider the following main-level program that realizes an unmapped base, then adds 200 button widgets to the previously-realized base:

```
time = SYSTIME(1)
b = WIDGET_BASE(/COLUMN, XSIZE=400, YSIZE=400, MAP=0)
WIDGET_CONTROL, b, /REALIZE
```



```
FOR i = 0, 200 DO button = WIDGET_BUTTON(b, VALUE=STRING(i))
WIDGET_CONTROL, b, /MAP
PRINT, 'time used: ', SYSTIME(1) - time
END
```

This program takes approximately 1.5 seconds to run on a Red Hat linux workstation. If the base had been mapped, the user would see the base “flashing” as each button was added to the base. Altering the example to use the UPDATE keyword reduces the execution time to approximately 0.1 seconds and eliminates the flashing:

```
time = SYSTIME(1)
b = WIDGET_BASE(/COLUMN, XSIZE=400, YSIZE=400, MAP=0)
WIDGET_CONTROL, b, /REALIZE, UPDATE=0
FOR i = 0, 200 DO button = WIDGET_BUTTON(b, VALUE=STRING(i))
WIDGET_CONTROL, b, /MAP, /UPDATE
PRINT, 'time used: ', SYSTIME(1) - time
END
```

Note

Do not attempt to resize a widget on the Windows platform while UPDATE is turned off. Doing so may prevent IDL from updating the screen properly.

Tips on Creating Widget Applications

The following are some ideas to keep in mind when writing widget applications in IDL.

- When writing new applications, decompose the problem into sub-problems and write reusable compound widgets to implement them. In this way, you will build a collection of reusable widget solutions to general problems instead of hard-to-modify, monolithic programs.
- Use the `GROUP_LEADER` keyword to `WIDGET_BASE` to define the relationships between parts of your application. Group leadership/membership relationships make it easy to group widgets appropriately for iconization, layering, and destruction.
- Use the `MBAR` keyword to `WIDGET_BASE` to create application-specific menubars. Use keyboard focus events to track which widget menu options should affect.
- Use existing compound widgets when possible. In particular, use the `CW_BGROUN` and `CW_PDMENU` compound widgets to create menus. These functions are easier to use than writing the menu code directly, and your intent will be more quickly understood by others reading your code.
- The many advantages of the `XMANAGER` procedure dictate that all widget programs should use it. There are few if any reasons to call the `WIDGET_EVENT` procedure directly.
- Use `CATCH` to handle any unanticipated errors. The `CATCH` branch can free any pointers, pixmaps, logical units, etc., to which the calling routine will not have access, and restore IDL session-wide settings like color tables and system variables that were locally modified.
- It can be difficult to write 100% portable widget code that looks good on all platforms, so let IDL do the layout for you when possible. If all else fails, it is possible to use the value of the `WIDGET_INFO` function to execute special-case code for each platform's user interface toolkit. It is desirable, however, to avoid large-scale special-case programming because this makes maintenance of the finished program more difficult. See [“Portability Issues”](#) below for additional suggestions.
- Use the `BITMAP` keyword to `WIDGET_BASE` to add a custom icon to your base widget (Windows platform only).

Portability Issues

Although IDL widgets are essentially the same on all supported platforms, there are some differences that can complicate writing applications that work well everywhere. The following hints should help you write such applications:

- Avoid specifying the absolute size and location of widgets whenever possible. (That is, avoid using the `XSIZE`, `YSIZE`, `XOFFSET`, and `YOFFSET` keywords.) The different user interface toolkits used by different platforms create widgets with slightly different sizes and layouts, so it is best to use bases that order their child widgets in rows or columns and stay away from explicit positioning. If you must use these keywords, try to isolate the affected widgets in a sub-base of the overall widget hierarchy to minimize the overall effect.
- When using a bitmap to specify button labels, be aware that some toolkits prefer certain sizes and give sub-optimal results with others.
- Try to place text, label, and list widgets in locations where their absolute size can vary without making the overall application look bad. The fonts used by the different toolkits have different physical sizes that can cause these widgets to have different proportions.

It is reasonably easy to write applications that will work in all environments without having to resort to much special-case programming. It is very helpful to have a machine running each environment available so that the design can be tested on each iteratively until a suitable layout is obtained.

Note

Also see [“Widgets in Cross-Platform Programs”](#) on page 172 for additional information.

Enhancing Widget Application Usability

Implementing features such as tabbing and keyboard accelerators into applications that require extensive user-interaction with widget elements can improve application usability. This allows power-users to quickly make selections and initiate actions using the keyboard instead of the mouse. See the following sections for details:

- “[Tabbing in Widget Applications](#)” in the following section describes the tabbing functionality and the differences in this functionality between platforms.
- “[Assigning Accelerators in Widget Applications](#)” on page 92 describes implementing keyboard accelerators for button widgets including menu items.

Tabbing in Widget Applications

Microsoft Windows and UNIX platforms support using the **Tab** key to navigate between IDL widgets (except draw widgets, label widgets and property sheet widgets on Windows). Under Windows, the `TAB_MODE` keyword determines how IDL widgets are affected by tabbing in an application. Under UNIX, the Motif library controls what widgets can receive and lose focus through tabbing. The `TAB_MODE` keyword is ignored when running a widget application on the UNIX platform.

Note

It is not possible to tab to disabled (`SENSITIVE=0`) or hidden (`MAP=0`) widgets.

The following table highlights other differences in tabbing functionality between the two platforms.

Widget	Description
WIDGET_BUTTON (Grouped, exclusive button widgets, also known as radio buttons)	<ul style="list-style-type: none"> • On Windows – radio buttons can receive and lose focus through tabbing as long as there is a selected button within the group. Use the arrow keys to change the selection within the group. • On UNIX — the Motif library controls tabbing functionality.
WIDGET_BUTTON (Grouped, non-exclusive button widgets also known as check boxes)	<ul style="list-style-type: none"> • On Windows — toggle buttons (or check boxes) can individually receive and lose focus through tabbing. Use the Space key to select or deselect a check box. • On UNIX — the Motif library controls tabbing functionality.
WIDGET_DROPLIST	<ul style="list-style-type: none"> • On Windows — droplist widgets can receive and lose focus through tabbing. • On UNIX — the Motif library controls tabbing functionality.
WIDGET_TABLE	<ul style="list-style-type: none"> • On Windows — table widgets can receive and lose focus through tabbing, but the focus is not clearly depicted. • On UNIX — the Motif library controls tabbing functionality.

Table 3-1: Tabbing Behavior in Windows and UNIX Widget Applications

Widget	Description
WIDGET_TEXT	<ul style="list-style-type: none"> • On Windows — text widgets can receive and lose focus through tabbing. When a text widget can lose focus via tabbing, keypress events are not generated for the Tab key and tab characters are not inserted into the text field. • On UNIX — tabbing behavior is controlled by the Motif library, and may vary from platform to platform. For single-line text widgets, the value of the TAB_MODE keyword is ignored, keypress events are not generated for the Tab key, and tab characters are not inserted into the text field. For multi-line text widgets, the behavior is the same as under Windows.
WIDGET_TREE	<ul style="list-style-type: none"> • On Windows — tree widgets can receive and lose focus through tabbing. Use the arrow keys to select higher or lower level nodes. • On UNIX — the Motif library controls tabbing functionality.

Table 3-1: Tabbing Behavior in Windows and UNIX Widget Applications (Continued)

Note

WIDGET_LABEL, WIDGET_PROPERTY SHEET, WIDGET_DRAW, and menu-related widgets do not support receiving or losing focus through tabbing. See the TAB_MODE keyword for each widget in the *IDL Reference Guide* for special behavior and navigation notes.

Defining Tabbing Behavior in a Windows Application

The TAB_MODE keyword provides control over tabbing behavior in a Windows application. This keyword controls navigation by specifying how a given widget should respond to the **Tab** key.

Allowable values are:

Value	Description
0	Disable navigation onto or off of the widget. This is the default unless the TAB_MODE has been set on a parent base. Child widgets automatically inherit the tab mode of the parent base as described in “ Inheriting the TAB_MODE Value ” on page 88.
1	Enable navigation onto and off of the widget.
2	Navigate only onto the widget.
3	Navigate only off of the widget.

Table 3-2: TAB_MODE Keyword Options

Note

Widgets including top level bases have a TAB_MODE value of zero by default.

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The TAB_MODE keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform. Tabbing behavior may vary significantly between UNIX platforms; do not rely on a particular behavior being duplicated on all UNIX systems.

Many compound widgets also support the TAB_MODE keyword. See each CW_* widget in the *IDL Reference Guide* for more information.

Navigation Among Widgets Using Tabbing

Navigation among widgets follows the widget hierarchy. Although it is not possible to specify a tab order, the widget tree hierarchy provides a natural progression among the widgets. Traversal is depth-first, meaning that once a widget receives focus through tabbing, additional tabbing will navigate through the interior nodes of the widget if possible before traversing to the next widget.

The TAB_MODE is either inherited from a parent base or explicitly set on a widget. However, to understand the effective range of a TAB_MODE setting, the TAB_MODE keyword value and the current focus must be considered.

- Setting `TAB_MODE` on a top level base — this setting is inherited by all lower level bases and child widgets on which `TAB_MODE` is not explicitly set.
- Setting `TAB_MODE` on an intermediate base — this setting is inherited by child widgets if `TAB_MODE` is not explicitly set on a widget. For example, if the top level base `TAB_MODE=0`, but the base associated with a group of buttons has a `TAB_MODE=1`, then when any of the buttons in the group is selected, tabbing will navigate among the buttons. If focus is anywhere other than on this base's elements, tabbing is disabled.
- Setting `TAB_MODE` on a widget — this setting affects tabbing capabilities only when the widget has focus. For example, if the parent base has a `TAB_MODE=1` (enabling tabbing), but a slider widget has a `TAB_MODE=3`, then the slider cannot receive focus through tabbing, it can only lose focus.

Note

Depressing the **Tab** key navigates down the widget hierarchy or to the right.
Depressing **Shift+Tab** navigates up the widget hierarchy or to the left.

Specifying and Inheriting `TAB_MODE`

The `TAB_MODE` keyword is allowed in most widget creation routines with the exception of `WIDGET_LABEL`, `WIDGET_PROPERTY SHEET`, and `WIDGET_DRAW`. The `TAB_MODE` keyword also *cannot* be explicitly set for the following widgets:

- Grouped, exclusive button widgets (radio buttons)
- Menu items or menu bases

Attempting to set `TAB_MODE` on these widgets will generate an error.

Inheriting the `TAB_MODE` Value

Tabbing behavior is inherited from a `WIDGET_BASE`. When `TAB_MODE` is set on a base widget, child widgets inherit the setting when they are created. This provides a quick way of enabling or disabling tabbing for all widgets belonging to a base. This is especially useful for widgets that do not directly support the `TAB_MODE` keyword. For example, attempting to set `TAB_MODE` for exclusive, grouped button widgets (radio buttons) will generate an error. Setting the tab mode on the parent base passes the specified tabbing functionality along to all widget children on that base. In the following code, the base (`base`) is defined as one which can receive and lose focus through tabbing. The child widgets (`b1`, `b2`, and `b3`) inherit this setting and will receive and lose focus through tabbing as well.

Note

See the following section for the complete, working example.

```

; Define a base for the radio buttons.
base = WIDGET_BASE(tlb, /COLUMN, /FRAME, /EXCLUSIVE, TAB_MODE = 1)

b1 = widget_button( base, $
    value = "MorphOpen" , UVALUE="Open")
b2 = widget_button( base, $
    value = "MorphClose" , UVALUE="Close")
b3 = widget_button( base, $
    value = "Dilate " , UVALUE="Dilate")

; Set button one as selected.
WIDGET_CONTROL, b1, /SET_BUTTON

```

Use `WIDGET_CONTROL` to make an initial selection within the group of radio buttons. This needs to be set before the group can receive focus through tabbing.

Note

For a child widget to receive focus through tabbing, the parent base must have a value of `TAB_MODE=1` (receive and lose focus) or `TAB_MODE=2` (only receive focus). A parent base with a `TAB_MODE=0` or `TAB_MODE=3` insulates child widgets from receiving focus. Only when focus is on the child widget would the child's individual `TAB_MODE` value be in effect.

Specifying TAB_MODE Values for Individual Widgets

The tab mode of the parent base is inherited by child widgets, but it is possible to control what widgets can receive or lose focus by specifying different `TAB_MODE` values on lower-level bases or individual widgets. For example, consider a top level base populated with a group of radio buttons, a group of check boxes, and a slider widget. The top level base has `TAB_MODE=1`, meaning that this base, and all widgets that inherit the setting from the base, will be able to receive and lose focus through tabbing. However, the `TAB_MODE` of the slider widget is explicitly set to 3 meaning that it can lose, but not receive focus. This excludes the widget from receiving focus when navigating the widget hierarchy. However, when focus is on a widget with a `TAB_MODE` keyword value of 3, and the **Tab** key is depressed, focus leaves the current widget and returns to the first widget that accepts focus through tabbing. The following simple example illustrates these concepts.

```

pro tabbing_example_event, event

; Return and print the uvalue of the widget with focus.
WIDGET_CONTROL, event.ID, GET_UVALUE = uvalue

```

```

PRINT, 'Event on: ', UVALUE

end

pro tabbing_example

; Create a top level base. Specify a tab mode that allows child
; widgets to receive and lose focus (TAB_MODE=1).
tlb = WIDGET_BASE(/COLUMN, TITLE = "Tabbing Example", $
    XPAD=0, YPAD=10, XOFFSET=25, YOFFSET=25, TAB_MODE=1)

; Create a base with radio buttons that inherits the ability
; to receive and lose focus through tabbing from parent tlb. This
; setting is also inherited by widget children of rbase.
rbase = WIDGET_BASE(tlb, /COLUMN, /FRAME, /EXCLUSIVE)
rb1 = WIDGET_BUTTON(rbase, VALUE = "MorphOpen", UVALUE = "Open")
rb2 = WIDGET_BUTTON(rbase, VALUE = "MorphClose", UVALUE = "Close")
rb3 = WIDGET_BUTTON(rbase, VALUE = "Dilate", UVALUE = "Dilate")

; Mark the first button as selected to enable tabbing to the
; group of radio buttons.
WIDGET_CONTROL, rb1, /SET_BUTTON

; Create a base with check boxes that inherits the ability to
; receive and lose focus through tabbing from tlb. This setting
; is also inherited by widget children of cbase.
cbase = WIDGET_BASE(tlb, /COLUMN, /FRAME, /NONEXCLUSIVE)
b1 = WIDGET_BUTTON(cbase, $
    VALUE = "Structuring Element: 3x3", UVALUE = "se3x3")
b2 = WIDGET_BUTTON(cbase, $
    VALUE = "Structuring Element: 5x5", UVALUE = "se5x5")

; Create a slider widget. Set the tab mode so that it can
; lose focus, but not receive focus through tabbing.
slider = WIDGET_SLIDER(tlb, UVALUE = 'slider', TAB_MODE=3)

; Draw the widgets and activate events.
WIDGET_CONTROL, tlb, /REALIZE
XMANAGER, 'tabbing_example', tlb, /NO_BLOCK

end

```

Save and run the above code. This results in a group of widgets similar to the following figure.

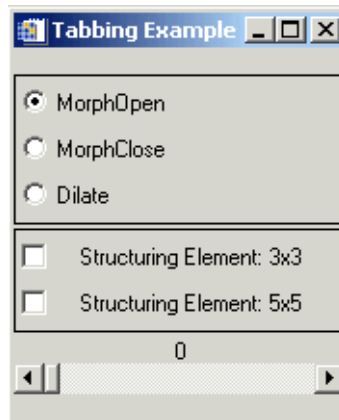


Figure 3-8: Navigating Widget Hierarchies Using Tabbing

Select a radio button or check box and then depress the **Tab** key to navigate between these widgets. With the mouse select and move the slider. Press the **Tab** key and the focus shifts to the group of radio buttons.

On Windows, use the arrow keys to navigate among the radio button options. Tab to the check boxes. Depress the **Space** key to select or deselect a check box. Use the **Tab** key to navigate through the check box options. Once the slider has focus, the arrow keys, Page Up and Page Down keys can be used to move the slider marker.

On UNIX, the `TAB_MODE` keyword is ignored. The arrow keys can be used to navigate between radio buttons, check boxes, and move the slider marker.

Modifying and Accessing the `TAB_MODE` Keyword

The `TAB_MODE` keyword value can be changed using `WIDGET_CONTROL` and queried using `WIDGET_INFO`. A change made to `TAB_MODE` using `WIDGET_CONTROL` affects only the widget for which the change is explicitly made. If changed on a widget base, the change is not propagated to the child widgets that have already been created. Use `WIDGET_CONTROL` to change the `TAB_MODE` value of any widget that supports the `TAB_MODE` keyword.

For example, if the `TAB_MODE` keyword value of the top level base is changed from 1 to 0 after child widgets have been created, tabbing will still be enabled for any applicable child bases or widgets when they have focus.

```
WIDGET_CONTROL, tlb, TAB_MODE=0
```

Use `WIDGET_INFO` to return any widget's `TAB_MODE` keyword value. For example, the following returns the keyword value of a slider widget, `slider`.

```
; Query the tabbing capabilities of a slider widget.
vtabmode = WIDGET_INFO(slider, /TAB_MODE)
print, vtabmode
```

Note

In widget applications on the UNIX platform, the Motif library controls what widgets are brought into and released from focus using tabbing. The `TAB_MODE` keyword value is always zero, and any attempt to change it is ignored when running a widget application on the UNIX platform. Tabbing behavior may vary significantly between UNIX platforms; do not rely on a particular behavior being duplicated on all UNIX systems.

Assigning Accelerators in Widget Applications

Keyboard accelerators allow the user to activate button widget events using keyboard key combinations instead of the mouse. On Windows platforms, accelerators can be defined for menu items and various types of `WIDGET_BUTTON`. Note, however, that:

- UNIX and Macintosh platforms support only menu item accelerators
- context menu items do not support accelerators on any platform

Note

Ordinarily, accelerators are processed before keyboard events reach a widget. This can cause button accelerators to steal keyboard events from widgets that have focus. If you find that this is an issue, see [“Disabling Button Widget Accelerators”](#) on page 97.

Successfully Implementing Keyboard Accelerators

The following tips should be kept in mind when adding accelerators to button widgets:

- Additional work is required to enable accelerators using the **Alt** key to work on Mac. To get the **Apple** (command) key to function as the **Alt** key, see [“Enabling Alt Key Accelerators on Macintosh”](#) on page 93.

- When an accelerator is implemented, it intercepts keyboard events before they are passed to the widgets. Widgets will never see keyboard events that are mapped to accelerators unless the accelerator is disabled as described in “Disabling Button Widget Accelerators” on page 97. For example, if **Ctrl+C** is mapped to a button that creates a contour plot, the key combination will no longer perform the copy function.

Note

Under Sun operating systems, the **Delete** key will not function as an accelerator if a widget has keyboard focus.


- Be mindful of the inherent operating system keyboard combinations when choosing your accelerators. For example, avoid a **Ctrl+Alt+Del** accelerator for an application run on the Windows platform.
- To support the greatest cross-platform portability, consider avoiding mapping function keys **F9** to **F12** for the following reasons:
 - Sun operating systems have the **F11** and **F12** keys mapped to SunF36 and SunF37. Attempting to reassign them can be problematic.
 - Mac OS 10.3 with Expose uses **F9**, **F10**, **F11**. Additionally, **Insert** and **Alt** may be unavailable.

Enabling Alt Key Accelerators on Macintosh

Two steps are required to enable accelerators that use the **Alt** key to work with the Macintosh **Apple** key (also known as the **Command** key):

1. Create a `.xmodmap` file in your home folder and add the following three lines to it:

```
clear mod1
clear mod2
add mod1 = Meta_L
```

When Apple’s X11 program starts, this file will automatically be read, and the Apple key will be mapped to the left meta key , which for IDL’s purposes is the **Alt** key. (Windows **Alt** key accelerators are mapped to the Macintosh **Apple** key, not the **Option (alt)** key.)

2. Run Apple’s X11 program and change its preferences. Under **Input** in the X11 Preferences dialog, make sure that the following two items are *unchecked*:
 - Follow system keyboard layout
 - Enable key equivalents under X11

Note

You must relaunch Apple's X11 program for these changes to take effect.

Performing these two steps will also have the benefit of making the IDL Workbench's keyboard shortcuts operate in the normal Macintosh fashion. Namely, pressing the **Apple** (⌘ key) in conjunction with X, C, and V will perform cut, copy and paste. The IDL Workbench's other shortcuts will also work. If you distribute your application to other Macintosh users, they too will need to have an appropriate .Xmodmap and correct X11 Preferences dialog settings in order for **Alt** key accelerators to work.

Specifying WIDGET_BUTTON Accelerators

The ACCELERATOR keyword assigns a key combination that activates a menu item or button event. The value of the keyword is a case-insensitive string that specifies zero or more modifier keys (**Ctrl**, **Shift**, or **Alt**) and one other key. (Mac users must take special steps to enable **Alt** key accelerators. See [“Enabling Alt Key Accelerators on Macintosh”](#) on page 93 for details.) If there is more than one item in the string, a “+” must separate them. For example:

```
base = WIDGET_BASE( t1b, /COLUMN, /FRAME )
bRun = WIDGET_BUTTON( base, VALUE = "Run", ACCELERATOR = "F5" )
bPause = WIDGET_BUTTON( base, VALUE = "Pause", $
    ACCELERATOR = "Ctrl+F5" )
bResume = WIDGET_BUTTON( base, VALUE = "Resume", $
    ACCELERATOR = "Ctrl+Shift+F5" )
```

The valid combinations are:

Accelerator Keys	Description
Ctrl , Shift , or Alt plus an alphanumeric key	A modifier key plus any alphabetic character, A-Z (which is case-insensitive), or a number, 0-9, creates a valid accelerator.
Ctrl , Shift , or Alt plus a number pad key	A modifier key plus any key on the number pad can be used as an accelerator. The NumLock key must be activated for any accelerator using number pad keys to function properly. Note - On Windows only, a keyboard accelerator using the Shift key and a number key on the number pad will not work.

Table 3-3: Valid ACCELERATOR Keyword Combinations

Accelerator Keys	Description
Ctrl, Shift, or Alt plus the BackSpace, Tab or Space key	These miscellaneous keys need a modifier key in the accelerator definition.
Navigation keys (Home, End, PageUp, PageDown, Up, Down, Left, Right)	The navigation keys do not require a modifier in the accelerator definition. Prior and PageUp are equivalent as are Next and PageDown . Up, Down, Left, and Right map to the arrow keys.
Function keys (F1 to F12)	Function keys do not need a modifier key in the accelerator definition. However, not all platforms support the use of all function keys as accelerators. See “Successfully Implementing Keyboard Accelerators” on page 92 for details.
Return, Escape, Insert, Del keys	These miscellaneous keys do not need a modifier key in the accelerator definition. You must specify Return in the accelerator definition to indicate the Enter key on Windows. You must specify Del in the accelerator definition to indicate the Delete key.

Table 3-3: Valid ACCELERATOR Keyword Combinations (Continued)

Note

Accelerators can be defined for menu items and other types of `WIDGET_BUTTON` on Windows. However, UNIX supports only menu item accelerators. Context menu items do not support accelerators on any platform.

When an accelerator is defined for a menu item, the `ACCELERATOR` keyword string is automatically displayed next to the menu item value. The `ACCELERATOR` keyword string is not included with a button value. Therefore, the `VALUE` keyword of a `WIDGET_BUTTON` that is not a menu item should also indicate the accelerator keyboard shortcut so that the user is aware of the option.

The following simple example creates a variety of `WIDGET_BUTTON` types with accelerators.

```
; AcceleratorExample.pro
; Example of the use of keyboard accelerators.

pro acceleratorexample_event, event
```

```

WIDGET_CONTROL, event.ID, GET_UVALUE = uvalue
PRINT, 'Event on: ', uvalue

IF ( uvalue EQ 'Quit' ) THEN BEGIN
    WIDGET_CONTROL, event.TOP, /DESTROY
END

end

pro AcceleratorExample

tlb = WIDGET_BASE( /ROW, $
    MBAR = mbar, TITLE = "Accelerator Example", $
    XPAD = 10, YPAD = 10, XOFFSET = 25, YOFFSET = 25 )

; Create a menu with accelerators. The accelerator string is
; automatically displayed along with the menu item text.
file = WIDGET_BUTTON( mbar, /MENU, $
    VALUE = "File" )

one = WIDGET_BUTTON( file, $
    VALUE = "One", UVALUE = "One", $
    ACCELERATOR = "Ctrl+1" )

two = WIDGET_BUTTON( file, $
    VALUE = "Two", UVALUE = "Two", $
    ACCELERATOR = "Ctrl+2" )

three = WIDGET_BUTTON( file, $
    VALUE = "Three", UVALUE = "Three", $
    ACCELERATOR = "Ctrl+3" )

quit = WIDGET_BUTTON( file, $
    VALUE = "Quit", UVALUE = "Quit", $
    ACCELERATOR = "Ctrl+Q" )

; Create a base with push buttons. Include the accelerator
; text in the button value so users are aware of it.
base = WIDGET_BASE( tlb, /COLUMN, /FRAME )

b1 = WIDGET_BUTTON( base, $
    VALUE = "Affirmative (Ctrl+Y)", UVALUE = "Yes", $
    ACCELERATOR = "Ctrl+Y" )

b2 = WIDGET_BUTTON( base, $
    VALUE = "Negative (Ctrl+N)", UVALUE = "No", $
    ACCELERATOR = "Ctrl+N" )

; Create a base with radio buttons.

```



```

base = WIDGET_BASE( tlb, /COLUMN, /FRAME, /EXCLUSIVE )

b1 = widget_button( base, $
    VALUE = "Owl (Ctrl+O)", UVALUE = "Owl", $
    ACCELERATOR = "Ctrl+O" )

b2 = WIDGET_BUTTON( base, $
    VALUE = "Emu (Shift+E)", UVALUE = "Emu", $
    ACCELERATOR = "Shift+E" )

b3 = WIDGET_BUTTON( base, $
    VALUE = "Bat (Alt+B)", UVALUE = "Bat", $
    ACCELERATOR = "Alt+B" )

; Create a base with check boxes.
base = WIDGET_BASE( tlb, /COLUMN, /FRAME, /NONEXCLUSIVE )

b1 = WIDGET_BUTTON( base, $
    VALUE = "Hello (F3)", UVALUE = "Hello", $
    ACCELERATOR = "F3" )

b2 = WIDGET_BUTTON( base, $
    VALUE = "Goodbye (F4)", UVALUE = "Goodbye", $
    ACCELERATOR = "F4" )

; Create the widgets and accept events.
WIDGET_CONTROL, tlb, /REALIZE
XMANAGER, 'acceleratorexample', tlb, /NO_BLOCK

end

```

Save and run the example. The Output Log window reports which button has been activated using the accelerator.

Note

Menu item accelerators are only operational when the menu is closed.

Disabling Button Widget Accelerators

Keyboard events are intercepted by the accelerators before they are passed along to widgets. This means that a widget will never see a keyboard event that maps to an accelerator. This can be resolved using one of the following methods:

- Use the `IGNORE_ACCELERATORS` keyword on those widgets that you want to receive keyboard input regardless of defined accelerators. This is the

recommended method. See “Using IGNORE_ACCELERATORS” on page 98 for details.

- Disable the accelerated item by programmatically desensitizing the item when the widget that you want to receive events gains focus, and resensitize the item when the widget loses focus. This allows the keystrokes that would ordinarily map to the accelerator to reach the desired widget when it has focus.

Using IGNORE_ACCELERATORS

The IGNORE_ACCELERATORS keyword is available on the following widgets:

- WIDGET_COMBOBOX
- WIDGET_DRAW
- WIDGET_PROPERTY SHEET
- WIDGET_TABLE
- WIDGET_TEXT

For each widget with a text area, accelerator overrides are active only when focus is on an editable text portion. (Accelerator overrides for draw widgets are active when the drawing area has focus.) For example, when the focus is on a table cell that cannot be edited, accelerators are still enabled.

Note

Depending on system hardware, the number of widgets that have accelerators, and the number of accelerators ignored, you may notice a slight performance penalty.

Set the IGNORE_ACCELERATORS equal to the text string of a single accelerator, an array containing multiple accelerator strings, or 1 (to ignore all accelerators).

Managing Accelerators Example

The following example shows various ways accelerators can be managed. This example creates several menu items with accelerators. Three text boxes either allow all accelerators, some accelerators or no accelerators to receive keyboard events. Additionally you can select a checkbox to desensitize the Delete menu item. When the menu item is desensitized, the accelerator never receives keyboard events.

```
PRO manage_accel, event
END

PRO quit_event, event
  WIDGET_CONTROL, event.top, /DESTROY
END
```

```

PRO menu_event, event
    PRINT, WIDGET_INFO( event.id, /UNAME )
END

PRO menu_sense_event, event
    deleteItem = WIDGET_INFO( event.top, FIND_BY_UNAME = "MenuDel" )
    WIDGET_CONTROL, deleteItem, SENSITIVE = event.select
END

pro manage_accel

; Create the top level base.
tlb = WIDGET_BASE( /COLUMN, MBAR = mbar, XSIZE = 250, /TAB_MODE)

; Build the menu bar.
edit= WIDGET_BUTTON( mbar, /MENU, VALUE = "Edit" )
menuDel = WIDGET_BUTTON( edit, VALUE = "Delete", $
    UNAME = "MenuDel", ACCELERATOR = "Del", $
    EVENT_PRO = "menu_event" )
menuCut = WIDGET_BUTTON( edit, VALUE = "Cut", $
    UNAME = "MenuCut", ACCELERATOR = "Ctrl+X", $
    EVENT_PRO = "menu_event" )
menuCopy = WIDGET_BUTTON( edit, VALUE = "Copy", $
    UNAME = "MenuCopy", ACCELERATOR = "Ctrl+C", $
    EVENT_PRO = "menu_event" )
menuPaste = WIDGET_BUTTON( edit, VALUE = "Paste", $
    UNAME = "MenuPaste", ACCELERATOR = "Ctrl+V", $
    EVENT_PRO = "menu_event" )
menuUndo = WIDGET_BUTTON( edit, VALUE = "Undo", $
    UNAME = "MenuUndo", ACCELERATOR = "Ctrl+Z", $
    EVENT_PRO = "menu_event" )
quit = WIDGET_BUTTON( edit, VALUE = "Quit", $
    ACCELERATOR = "Ctrl+Q", EVENT_PRO = "quit_event" )

; Add text boxes with various levels of disabled accelerators.
text1 = WIDGET_TEXT( tlb, /EDITABLE, $
    VALUE = "Doesn't use IGNORE_ACCELERATORS." )
text2 = WIDGET_TEXT( tlb, /EDITABLE, $
    VALUE = "Receives Delete key and Ctrl+C combinations.", $
    IGNORE_ACCELERATORS = [ "Del", "Ctrl+C" ] )
text3 = WIDGET_TEXT( tlb, /EDITABLE, $
    VALUE = "Receives all accelerator key combinations.", $
    IGNORE_ACCELERATORS = 1)

; Add a check box to desensitize the Delete menu item.
base2 = WIDGET_BASE( tlb, /FRAME, /NONEXCLUSIVE )
check1 = WIDGET_BUTTON( base2, VALUE = "Menu DEL sensitive", $
    EVENT_PRO = "menu_sense_event" )

```

```
WIDGET_CONTROL, check1, SET_BUTTON = WIDGET_INFO ( menuDel, $
    /SENSITIVE )

; Draw the widget.
WIDGET_CONTROL, tlb, /REALIZE
XMANAGER, "manage_accel", tlb, /NO_BLOCK

END
```

Compile and run the example. Try highlighting and deleting, or copying and pasting text in each textbox using accelerators defined in the **Edit** menu. All keyboard events are ineffective (stolen by the accelerators) when the first textbox has focus. The second textbox receives only copy and delete keyboard combinations. The third textbox receives all accelerators. When the delete menu item is desensitized, the **Delete** key can delete text from all textboxes. The IDL Output Log window prints the name of any menu item that is activated using an accelerator.



Chapter 4

Using Widget Buttons

The following topics are covered in this chapter:

Using Button Widgets	102	Tooltips	106
Bitmap Button Labels	103	Exclusive and Non-Exclusive Buttons ...	107

Using Button Widgets

Button widgets allow users to respond to “yes-or-no” type questions via the widget interface. While button widgets are generally fairly simple to understand and use, there are numerous options that allow you to fine-tune the appearance and behavior of buttons in your interface. This section discusses some useful ideas and techniques for using button widgets. See “[WIDGET_BUTTON](#)” (*IDL Reference Guide*) for a complete description of the function used to create button widgets.

This section discusses the following topics:

- “[Bitmap Button Labels](#)” on page 103
- “[Tooltips](#)” on page 106
- “[Exclusive and Non-Exclusive Buttons](#)” on page 107

Note

Menus also make use of button widgets. See “[Creating Menus](#)” on page 64 for more information.

Bitmap Button Labels

In addition to setting the `VALUE` of a button widget to a text string, you can use a bitmap image as the label for the button. To use a bitmap image, set `VALUE` to one of the following:

- The path to a bitmap image file, if the `BITMAP` keyword is also specified.
- An $n \times m$ byte array converted to a bitmap byte array using the `CVTTOBM` function, which displays as a black-and-white bitmap image.
- An $n \times m \times 3$ byte array, which displays as a 24-bit color bitmap image.

The following sections describe the process of creating bitmap files, black-and-white arrays, and color arrays for use as bitmap button labels.

Creating Bitmap Files for Buttons

You can produce appropriate bitmap files (for use with the `BITMAP` keyword to `WIDGET_BUTTON`) using any bitmap editor available on your operating system. Be sure to save the file as a `.bmp` file.

Transparent Bitmaps

For 16- and 256-color bitmaps included using the `BITMAP` keyword, IDL uses the color of the pixel in the lower left corner as the transparent color. All pixels of this color become transparent, allowing the button color to show through. This allows you to use bitmaps that do not appear to be rectangular. For 24-bit bitmaps, there is no transparent pixel.

If you have a 16- or 256-color rectangular bitmap and you want to maintain the rectangular shape of a bitmap, you can either draw a border of a different color around the bitmap (making sure that the lower left pixel is a different color from the background you want to maintain) or save the bitmap as a 24-bit (TrueColor) image. If your bitmap also contains text, make sure the border you draw is a different color than the text, otherwise the text color will become transparent.

Note on 8-bit X Windows Displays

Displaying bitmap buttons on 8-bit color X Windows displays may require using additional X colormap colors to allocate colors used by the bitmaps. If the required colormap colors are not available, the button bitmap may not display properly.

Creating Black-and-White Bitmap Arrays for Buttons

You can produce appropriate black-and-white bitmap arrays in IDL in the following ways:

- Create a black and white bitmap using an external bitmap editor, and read it into an IDL byte array using the appropriate procedure (`READ_BMP`, `READ_JPEG`, etc.) and convert the byte array to a bitmap byte array using the `CVTTOBM` function.
- On an X-Window system, use the X11 bitmap utility to create a black and white bitmap byte array and read it in to IDL using the `READ_X11_BITMAP` routine.
- Create a black and white bitmap using the `XBM_EDIT` procedure. This procedure offers several alternatives for the form of the final bitmap.
- Create an $n \times m$ byte array using the `BYTARR` function and modify array elements using array operations. Use `CVTTOBM` to convert the array to a bitmap byte array.

Creating Color Bitmap Arrays for Buttons

You can produce appropriate color bitmap arrays in IDL in the following ways:

- Create a 24-bit color image using an external bitmap editor, and read it into an IDL byte array using the appropriate procedure (`READ_BMP`, `READ_JPEG`, etc.). Remember that the image array must be interleaved by plane ($n \times m \times 3$), with the planes in the order red, green, blue. Note that image files created by image editors are often interleaved by pixel rather than by plane; use the `TRANSPOSE` function to reformat the array.

For example, if you read a 24-bit color image into an array using the `READ_BMP` function, the resulting array will be interleaved by pixel (with dimensions $3 \times n \times m$), with planes in the order blue, green, red. To create an array in the proper format for use as a button bitmap, use the following IDL commands:

```
button_image = READ_BMP('bitmap_file.bmp', /RGB)
button_image = TRANSPOSE(button_image, [1,2,0])
...
button = WIDGET_BUTTON(base, VALUE=button_image)
```

Here, the `RGB` keyword to `READ_BMP` reorders the color planes to be in the order red, green, blue; the call to `TRANSPOSE` puts the array in the proper format for use in a bitmap button.

- Create an $n \times m \times 3$ byte array using the `BYTARR` function and modify the array elements using array operations.

Although IDL places no restriction on the size of bitmap allowed, the various toolkits may prefer certain sizes.

Tooltips

You can specify a “tooltip” — a short text string that will appear when the mouse pointer hovers over a button widget — by specifying the string as the value of the `TOOLTIP` keyword to `WIDGET_BUTTON`.

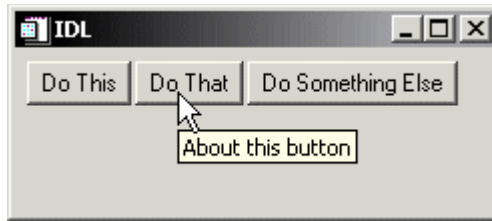


Figure 4-1: A tool tip.

Note

Tooltips cannot be created for menu sub-items. The topmost button of a pulldown menu can, however, have a tooltip.

Exclusive and Non-Exclusive Buttons

By default, when a user clicks on a button widget, the button appears to be depressed while the user holds down the mouse button, but the button returns to the undepressed appearance when the user releases the mouse button. While such “normal” buttons visually reflect the state of the button (depressed or undepressed), normal buttons are used to gather a single piece of information: whether the user clicked on the button or not.

Buttons placed into exclusive or non-exclusive bases (created via the `EXCLUSIVE` or `NONEXCLUSIVE` keywords to `WIDGET_BASE` procedure) are created as two-state “toggle” buttons—radio buttons (exclusive) or checkboxes (nonexclusive). Visually, when a user clicks on an exclusive or nonexclusive button, it remains in the depressed state, either until the user clicks on it again or (in the case of exclusive buttons) until another button in the group is depressed. Buttons that toggle in this manner can be used to gather information about a quantity that has two possible states.

Exclusive and nonexclusive buttons differ in the following way:

- If a base is created with the `EXCLUSIVE` keyword, only one button on the base can be selected at a given time. If one button is selected and another button pressed, the first button becomes unselected.
- If a base is created with the `NONEXCLUSIVE` keyword, any number of buttons can be selected at a given time. Pressing one button has no effect on the selected/unselected state of other buttons on the base.

Exclusive and nonexclusive buttons take on different appearances, depending on the type of button and on the windowing toolkit in use (Microsoft Windows or Motif).

Often, it is easier to create groups of buttons (normal, exclusive, or nonexclusive) using the `CW_BGROU`P compound widget than it is to program them yourself from base and button widgets and manage the events from each button individually. See [“Button Groups”](#) on page 64 and [“CW_BGROU”](#) (*IDL Reference Guide*) for additional information on using button groups.



Chapter 5

Using Draw Widgets

The following topics are covered in this chapter:

Using Draw Widgets	110	Context Events in Draw Widgets	117
Using Direct Graphics in Draw Widgets ..	111	Draw Widget Example	118
Using Object Graphics in Draw Widgets .	112	Accessing Draw Widget Events	119
Scrolling Draw Widgets	113	Implementing Drag and Drop Functionality ..	121

Using Draw Widgets

Draw widgets are graphics windows that appear as part of a widget hierarchy rather than appearing as an independent window. Like other graphics windows, draw widgets can be created to use either Direct or Object graphics. (See [Chapter 5, “Graphic Display Essentials”](#) (*Using IDL*) for a discussion of IDL’s two graphics modes.) Draw widgets allow designers of IDL graphical user interfaces to take advantage of the full power of IDL graphics in their displays. See [“WIDGET_DRAW”](#) (*IDL Reference Guide*) for a complete description of the function used to create draw widgets.

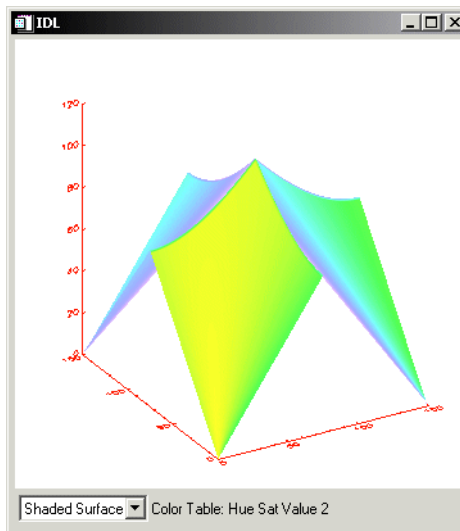


Figure 5-1: An IDL Draw Widget Displaying a Shaded Surface

Using Direct Graphics in Draw Widgets

By default, draw widgets use IDL Direct graphics. (To create a draw widget that uses Object graphics, set the `GRAPHICS_LEVEL` keyword to `WIDGET_DRAW` equal to two; see “[Using Object Graphics in Draw Widgets](#)” on page 112.) Once created, draw widgets using Direct graphics are used in the same way as standard Direct graphics windows created using the `WINDOW` procedure.

All IDL Direct graphics windows are referred to by a window number. Unlike windows created by the `WINDOW` procedure, the window number of a Direct graphics draw widget cannot be assigned by the user. In addition, the window number of a draw widget is not assigned until the draw widget is actually realized, and thus cannot be returned by `WIDGET_DRAW` when the widget is created. Instead, you must use the `WIDGET_CONTROL` procedure to retrieve the window number, which is stored in the *value* of the draw widget, *after* the widget has been realized.

Unlike normal graphics windows, creating a draw widget does not cause the current graphics window to change to the new widget. You must use the `WSET` procedure to explicitly make the draw widget the current graphics window. The following IDL statements demonstrate the required steps:

```
;Create a base widget.
base = WIDGET_BASE()

;Attach a 256 x 256 draw widget.
draw = WIDGET_DRAW(base, XSIZE = 256, YSIZE = 256)

;Realize the widgets.
WIDGET_CONTROL, /REALIZE, base

;Obtain the window index.
WIDGET_CONTROL, draw, GET_VALUE = index

;Set the new widget to be the current graphics window
WSET, index
```

If you attempt to get the value of a draw widget before the widget has been realized, `WIDGET_CONTROL` returns the value -1, which is not a valid index.

Using Object Graphics in Draw Widgets

To create a draw widget that uses Object graphics, set the `GRAPHICS_LEVEL` keyword to `WIDGET_DRAW` equal to two. Once created, draw widgets using Object graphics are used in the same way as standard `IDLgrWindow` objects.

All IDL Object graphics windows (that is, `IDLgrWindow` objects) are referred to by an object reference. Since you do not explicitly create the `IDLgrWindow` object used in a draw widget, you must retrieve the object reference by using the `WIDGET_CONTROL` procedure to get the *value* of the draw widget. As with Direct graphics draw widgets, the window object is not created—and thus the object reference cannot be retrieved—until after the draw widget is realized. If you attempt to retrieve the object reference for a draw widget's `IDLgrWindow` object before the draw widget is realized, IDL returns a null object.

Scrolling Draw Widgets

Another difference between a draw widget and either a graphics window created with the WINDOW procedure or an IDLgrWindow object is that draw widgets can include scroll bars. Setting the APP_SCROLL keyword or the SCROLL keyword to the WIDGET_DRAW function causes scrollbars to be attached to the drawing widget, which allows the user to view images or graphics larger than the visible area.

Differences Between SCROLL and APP_SCROLL

The amount of memory used by a draw widget is directly related to the size of the drawable area of the widget. If a draw widget does not have scroll bars, the entire drawable area is viewable. In this case, the size of the drawable area is controlled by the XSIZE and YSIZE keywords to WIDGET_DRAW.

With the addition of scroll bars, it is possible to display an image that is larger than the viewable area (the *viewport*) of the draw widget. IDL provides two options for dealing with images larger than the viewport:

1. Create the draw widget using the SCROLL keyword. This method creates a draw widget whose drawable area is specified by the XSIZE and YSIZE keywords, and whose viewable area is specified by the X_SCROLL_SIZE and Y_SCROLL_SIZE keywords. Since the entire image is kept in memory, IDL can display the appropriate portions automatically when the scroll bars are adjusted.
2. Create the draw widget using the APP_SCROLL keyword. This method creates a draw widget whose drawable area is the same size as its viewable area (specified by the X_SCROLL_SIZE and Y_SCROLL_SIZE keywords), but which can be different from the *virtual drawable area* (specified by the XSIZE and YSIZE keywords) that is equal to the full size of the image. In this case, only the portion of the image that is currently visible in the viewport is kept in memory; the IDL programmer must use viewport events to determine when the scroll bars have been adjusted and display the appropriate portion of the full image.

The concept of a virtual drawable area allows you to display portions of very large images in a draw widget without the need for enough memory to display the entire image. The price for this facility is the need to manually handle display of the correct portion of the image in an event-handling routine.

Example Using SCROLL

The following code creates a simple scrollable draw widget and displays an image.

Note

This example is included in the file `draw_scroll.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution. You can either open the file in an IDL editor window and compile and run the code using items on the **Run** menu, or simply enter

```
draw_scroll
```

at the IDL command prompt. See “[Running the Example Code](#)” on page 15 if IDL does not run the program as expected. You may need to enter `DEVICE, RETAIN=2` at the IDL command prompt before running this example.

```

; Event-handler routine. Does nothing in this example.
PRO draw_scroll_event, ev

END

; Widget creation routine.
PRO draw_scroll

; Read an image for use in the example.
READ_JPEG, FILEPATH('muscle.jpg', $
    SUBDIR=['examples', 'data']), image

; Create the base widget.
base = WIDGET_BASE()

; Create the draw widget. The size of the viewport is set to
; 200x200 pixels, but the size of the drawable area is
; set equal to the dimensions of the image array using the
; XSIZE and YSIZE keywords.
draw = WIDGET_DRAW(base, X_SCROLL_SIZE=200, Y_SCROLL_SIZE=200, $
    XSIZE=(SIZE(image))[1], YSIZE=(SIZE(image))[2], /SCROLL)

; Realize the widgets.
WIDGET_CONTROL, base, /REALIZE

; Retrieve the window ID from the draw widget.
WIDGET_CONTROL, draw, GET_VALUE=drawID

; Set the draw widget as the current drawable area.
WSET, drawID

; Load the image.
TVSCL, image

```

```

; Call XMANAGER to manage the widgets.
XMANAGER, 'draw_scroll', base, /NO_BLOCK

END

```

In this example, the drawable area created for the draw widget is the full size of the displayed image. Since IDL handles the display of the image as the scroll bars are adjusted, no event-handling is necessary to update the display.

Example Using APP_SCROLL

We can easily rework the previous example to use the APP_SCROLL keyword rather than the SCROLL keyword. Using APP_SCROLL has the following consequences:

1. IDL no longer automatically displays the appropriate portion of the image when the scroll bars are adjusted. As a result, we must add code to our event-handling procedure to check for the viewport event and display the appropriate part of the image. Here is the new event-handler routine:

```

; Event-handler routine.
PRO draw_app_scroll_event, ev

COMMON app_scr_ex, image

IF (ev.TYPE EQ 3) THEN TVSCL, image, 0-ev.X, 0-ev.Y

END

```

First, notice that since we need access to the image array in both the widget creation routine and the event handler, we place the array in a COMMON block. This is appropriate since the image data itself is not altered by the widget application.

Second, we check the TYPE field of the event structure to see if it is equal to 3, which is the code for a viewport event. If it is, we use the values of the X and Y fields of the event structure as the Position arguments to the TVSCL routine to display the appropriate portion of the image array.

2. We must add the COMMON block to the widget creation routine.
3. We change the call to WIDGET_DRAW to include the APP_SCROLL keyword rather than the SCROLL keyword. In this context, the values of the XSIZE and YSIZE keywords are interpreted as the size of the *virtual* drawable area, rather than the actual drawable area.

Example Code

The modified example is included in the file `draw_app_scroll.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution. Run this example procedure by entering `draw_app_scroll` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT draw_app_scroll.pro`.

On the surface the two examples appear identical. The difference is that the example using `APP_SCROLL` uses only the memory necessary to create the smaller drawable area described by the size of the viewport, whereas the example using `SCROLL` uses the memory necessary to create the full drawable area described by the `XSIZE` and `YSIZE` keywords. While the example image is not so large that this makes much difference, if the image contained several hundred million pixels rather than a few hundred thousand, the memory saving could be significant.

Context Events in Draw Widgets

The `WIDGET_DRAW` function does not have a `CONTEXT_EVENTS` keyword to specify that context menu events be generated when the user clicks the right mouse button over a drawable area. Instead, the event structure generated by draw widgets when the `BUTTON_EVENTS` keyword is set includes the `PRESS` and `RELEASE` fields, both of which contain information regarding which mouse button was pressed.

See [“Context-Sensitive Menus”](#) on page 69 for techniques used to simulate the generation of context menu events with draw widgets.

Draw Widget Example

The following example program creates a small widget application consisting of a draw widget and a droplist menu. One of three plots is displayed in the draw widget depending on the selection made from the droplist. To add to dynamic behavior, we will use timer events to change the color table used in the draw window every three seconds.

Example Code

This example is included in the file `draw_widget_example.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution. Run this example procedure by entering `draw_widget_example` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT draw_widget_example.pro`. See [“Running the Example Code”](#) on page 15 if IDL does not run the program as expected.

This procedure checks the type of event structure returned. See [“Identifying Widget Type from an Event”](#) on page 57 for more on identifying widget types from returned event structures.

The intent of this example is to demonstrate the use of draw widgets, menus, and timer events with a minimum of other complicating issues. However, it is easy to imagine applications wherein a graphics window containing a plot or some other information is updated periodically by a timer. The method used here can be easily applied to more realistic situations.

Accessing Draw Widget Events

To go beyond merely displaying an image in a draw widget and allow the user to interact in some way with the displayed image, you must configure the draw widget to generate either *button*, *motion*, *wheel*, or *keyboard* events:

- *Button events* are enabled by setting the `BUTTON_EVENTS` keyword to `WIDGET_DRAW`. Once enabled, button events are generated when the user clicks on the draw widget.
- *Motion events* are enabled by setting the `MOTION_EVENTS` keyword to `WIDGET_DRAW`. Once enabled, motion events are generated whenever the cursor moves over the draw widget.
- *Wheel events* are enabled by setting the `WHEEL_EVENTS` keyword to `WIDGET_DRAW`. Once enabled, wheel events are generated when the draw widget has focus and the user rolls the scroll wheel.

Note

Wheel events are enabled only under Microsoft Windows.

- *Keyboard events* are enabled by setting the `KEYBOARD_EVENTS` keyword to `WIDGET_DRAW`. Once enabled, events are generated when the draw widget has focus and a keyboard key is pressed.

The following example uses motion events to update the values of several label widgets as the mouse cursor moves over an image in a draw widget. This and several other features are discussed in the section following the code.

Example Code

See the file `draw_widget_data.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution for the example code. Run this example procedure by entering `draw_widget_data` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT draw_widget_data.pro`. See “[Running the Example Code](#)” on page 15 if IDL does not run the program as expected. You may need to enter `DEVICE, DECOMPOSED=1` at the IDL command prompt before running this example.

The following things about this example are worth noting:

- Since we use the image data in both the widget creation routine (where we display the image) and the event-handler routine (where we retrieve the value of the data point under the cursor), we need access to the variable that holds the

image in both places. We could pass the entire image array from the creation routine to the event-handler in the `stash` structure, but since the image could be large, we choose to pass a *pointer* to the image instead. This means we must dereference the pointer variable every time we need to use the image data. For more information on pointers and how to dereference them, see [Chapter 17, “Pointers”](#).

- In this example we have set the `MOTION_EVENTS` keyword to `WIDGET_DRAW`; this causes events to be generated continuously as the cursor moves across the draw widget. We could have set the `BUTTON_EVENTS` keyword instead; this would force the user to click the draw widget in order to update the text fields.

Implementing Drag and Drop Functionality

In IDL versions 6.3 and later, you can create applications that allow users to drag tree nodes from a tree widget to a draw widget. Drag and drop functionality is not enabled by default. When creating an IDL application that incorporates both a tree widget and a draw widget you can enable drag and drop behavior to drag values from the tree widget to the draw widget. This section discusses the steps necessary to implement drag and drop functionality in your application.

Implementing drag and drop functionality in your application entails three steps:

1. **Making Nodes Draggable.** You must explicitly specify that a node or group of nodes in the tree widget can be dragged. See “[Dragging and Dropping Tree Nodes](#)” on page 193 for details.
2. **Responding to Drag Notifications (Callbacks).** When the user drags a tree node onto a draw widget, IDL generates a *notification*, which is passed to a *callback function*. In most cases, you can use the default callback function, but you can create your own callback function to handle special or complex situations. Drag notifications allow you to control if and where drops are allowed.
3. **Responding to Drop Events.** When the user releases the mouse button to drop the selected nodes, IDL generates a *drop event*. You can use the information contained in the drop event structure to perform an operation, such as loading an image or other visualization in the draw widget.

Responding to Drag Notifications (Callbacks)

When the user drags a group of selected nodes over a draw widget, IDL automatically calls the routine defined as the *drag notification callback* for the draw widget. The purpose of the drag notification callback is to provide the widget system with information about where dragged nodes can be dropped, allowing it to change the cursor display to indicate to the user whether nodes can be dropped at the current position. You, as an IDL application programmer, cannot respond to the value returned by the drag notification callback directly, but you can choose to specify your own version of the callback function to override the default behavior. Drag notification callbacks are specified via the `DRAG_NOTIFY` keyword to `WIDGET_DRAW`, or the `SET_DRAG_NOTIFY` keyword to `WIDGET_CONTROL`.

Drag notifications are also generated when the state of a drag modifier key changes (either up or down). If you override the default drag notification callback, you can use this information to update the drag cursor with a plus symbol (+).

If no callback is defined for the draw widget, the default callback will be used.

Drag Notification Callback Return Values

The drag notification callback function returns an integer value calculated by ORing the following values together:

Value	Meaning
0	User cannot drop
1	User can drop onto
2	Show the plus indicator

Table 5-1: Drag Notification Callback Return Values

For example, if the callback returns the value 3, the user can drop onto the draw widget and the plus indicator will be displayed.

The Default Drag Notification Callback

The default drag notification callback function is used if no function is specified for the draw widget. The default callback returns 0 if drop events are not enabled (`DROP_EVENTS=0`) and 1 otherwise.

Writing Custom Drag Notification Callbacks

In most cases, the default drag notification callback should be adequate for an application that allows the user to drop tree nodes onto a draw widget. If it proves inadequate, however, you can create a custom callback to perform extra processing.

The drag notification callback routine has the following signature:

```
FUNCTION Callback_Function_Name, Destination, Source, $
    X, Y, Modifiers, Default
```

where

- *Callback_Function_Name* is the name of the callback function. This value is specified as the value of the `DRAG_NOTIFY` keyword.
- *Destination* is the widget ID of the draw widget over which the item is dragged.
- *Source* is the widget ID of the source tree, from which a list of widget IDs representing the list of selected nodes can be retrieved using the `TREE_SELECT` or `TREE_DRAG_SELECT` keywords to `WIDGET_INFO`.

- *X* is the position to the right of the lower left corner of the drawable area, in device coordinates (pixels).
- *Y* is the position above the lower left corner of the drawable area, in device coordinates (pixels).
- *Modifiers* indicates the state of the modifier keys. The widget system generates them by ORing the following values together for the depressed keys:

Bitmask	Modifier Key
1	Shift
2	Control
4	Caps Lock
8	Alt

Table 5-2: Bitmask and Corresponding Key

Note

For UNIX, the **Alt** key is the currently mapped MOD1 key.

- *Default* is the value that the default callback would have returned. A common usage is to have the callback return its value after modifying it to show the + indicator.

The return value should indicate where a drop is allowed to take place relative to the destination widget and whether the “+” symbol should appear with the drag cursor, as described in [Table 5-1](#). For additional information on writing drag notification callbacks, see “[Dragging and Dropping Tree Nodes](#)” on page 193.

Responding to Drop Events

When the user releases the mouse button over a valid drop target (that is, when the `DROP_EVENTS` keyword to `WIDGET_DRAW` has been set), a `WIDGET_DROP` event is generated. Your application’s event handler should recognize this drop event and perform some action.

The drop event’s information is contained in a `WIDGET_DROP` structure. (See [Drop Events](#) in the reference section for `WIDGET_DRAW` for a full definition of the `WIDGET_DROP` structure.) The important components of the structure when responding to drop events are:

- **ID** — The widget ID of the destination node.
- **DRAG_ID** — The widget ID of the source tree widget. The selected nodes of this tree are the nodes that are being dragged. You can use the `TREE_SELECT` keyword to `WIDGET_INFO` along with this widget ID to retrieve the list of selected nodes.
- **X** and **Y** — The drop position relative to the lower left corner of the drawable area.
- **MODIFIERS** — An integer representing the state of the modifier keys, calculated by ORing together the values shown in [Table 5-2](#). On some platforms it is common for the **Ctrl** key to be used as the copy key, with simple move operations being performed when **Ctrl** is not pressed.

Draw Widget Drag and Drop Example

The IDL distribution contains an example that contains a tree widget representing various image files and a draw widget onto which the tree nodes can be dragged to display the images.

Example Code

The draw widget drag and drop example is included in the file `drag_and_drop_draw.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution. Run this example procedure by entering `drag_and_drop_draw` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT drag_and_drop_draw.pro`.



Chapter 6

Using Property Sheet Widgets

The following topics are covered in this chapter:

Using Property Sheet Widgets	126	User-defined Properties	133
Registering Properties	127	Property Sheet Sizing	134
Selecting Properties	128	Property Sheet Example	136
Changing Properties	131	Multiple Properties Example	150

Using Property Sheet Widgets

The purpose of a property sheet (created with `WIDGET_PROPERTY SHEET`) is to enable the user to view and edit the properties of an object subclassed from the `IDLitComponent` class. (All `IDLit*` objects and most `IDLgr*` subclass from the `IDLitComponent` class.)

For example, a user may have rendered data as a surface. Using IDL's `iSurface` tool, the user can select the surface and bring up a property sheet that lists all of the surface's properties, including color, shading method, etc. To change the color, the user can go to the property sheet, select the color property, bring up the color picker, and select a new color. The name of the changed property is placed into an IDL event. It is in the processing of this event that the object is updated. An existing property sheet can be assigned a new component, which causes it to reload with the new list of properties and their values.

The topics in this chapter show how to use the property sheet widget with the `iTool`'s paradigm.

Registering Properties

In order for a property associated with a component object to be included in the property sheet for that component, the property must be *registered*. The property registration mechanism accomplishes several things:

- It allows you to expose as many or as few of the properties of an underlying object as you choose.
- It allows you to add user-defined properties to existing objects, and expose those new properties to users of your application.

Groups of properties of graphical atomic objects can be registered by setting their `REGISTER_PROPERTIES` properties to `True` when the object is initialized. See the property tables for each graphical atomic object in the *IDL Reference Guide*.

Selecting Properties

A property sheet consists of rows and columns. The left-most column identifies the properties, and the other column or columns identify the property values of one or more objects (also known as components). A select event is generated whenever a cell containing a property name or a property value is selected by left-clicking on it using the mouse. When a single property value is clicked on, the associated property name appears indented. Only a single property value can be selected at one time. However, when the `MULTIPLE_PROPERTIES` keyword is set, multiple properties can be selected in a property sheet using the **Ctrl** key to make nonadjacent selections or using the **Shift** key to make adjacent selections.

Note

Setting the `EDITABLE` keyword to 0 (zero) allows the user to select, but not modify properties. See “[WIDGET_PROPERTY SHEET](#)” (*IDL Reference Guide*) for details.

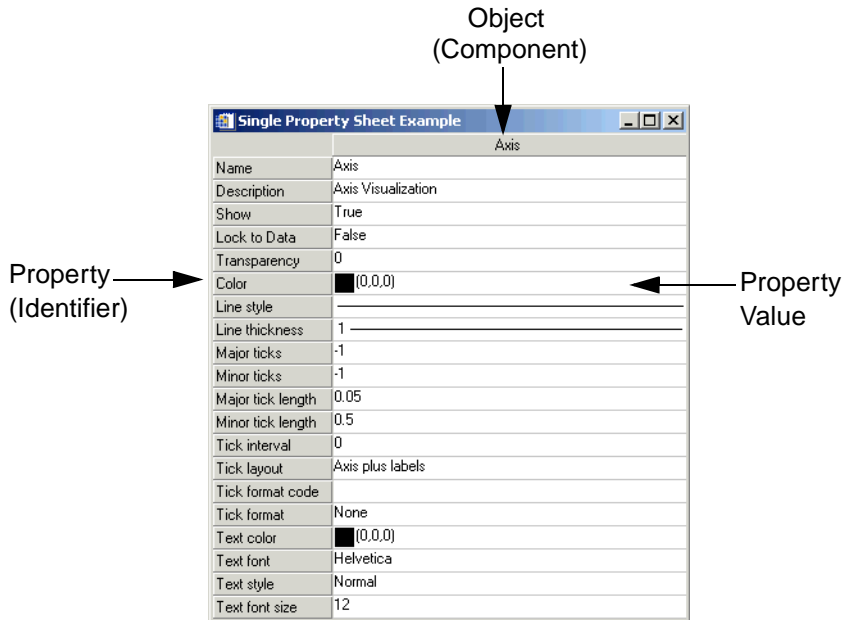


Figure 6-1: Property Sheet Selection

When the property sheet is initially realized, no properties are selected by default. However a single property or multiple properties can be selected programmatically using the `PROPERTY SHEET_SETSELECTED` keyword to the `WIDGET_CONTROL` procedure.

Set the `PROPERTY SHEET_SETSELECTED` keyword to a string or an array of strings identifying the properties to appear selected. The strings should match valid property identifiers. When this keyword is set to an empty string or an array that contains only an empty string, it clears all property selections. For example, the following code pre-selects two properties in a property sheet:

```
; Create the property sheet.
oComp = OBJ_NEW('IDLitVisAxis')
wPropAxis = WIDGET_PROPERTY SHEET(base, VALUE = oComp, $
    EVENT_PRO = 'PropertyEvent', UNAME = 'PropSheet', $
    /MULTIPLE_PROPERTIES)

; Pre-select the color and transparency properties of
; axis component.
WIDGET_CONTROL, wPropAxis,
    PROPERTY SHEET_SETSELECTED=['Color', 'Transparency']
```

Accessing Property Sheet Selection Events

The event structure (`WIDGET_PROPSHEET_SELECT`) provided when selection occurs contains a `COMPONENT` tag, an `IDENTIFIER` tag, and a `NSELECTED` tag.

```
{WIDGET_PROPSHEET_SELECT, ID:0L, TOP:0L, HANDLER:0L, TYPE:0L,
    COMPONENT:OBJREF, IDENTIFIER:"", NSELECTED:0L }
```

The `COMPONENT` tag is a reference to the object associated with the selected property value. When multiple objects (also known as components) are associated with the property sheet, this member indicates which one object had one of its property values selected. If a property (instead of a property value) is selected, the `COMPONENT` tag always contains an object reference to the first object, even if there are multiple objects in the property sheet. The `IDENTIFIER` tag uniquely identifies the property. This identifier is unique among all of the component's properties. The component and identifier can be used to obtain the value of the selected property:

```
isDefined = event.component-> $
    GetPropertyByIdentifier(event.identifier, value)
```

where `event` is the event structure, `isDefined` is a 1 if the value is defined (0, otherwise), and `value` receives the property's value.

The `NSELECTED` tag returns the number of currently selected properties. When more than a single property is selected, the `IDENTIFIER` field holds the identifier of the first item selected. This is not the first item selected with the mouse, but the first item encountered in the property sheet definition among those which are selected. The `NSELECTED` tag is equivalent to calling `WIDGET_INFO` with the `/PROPERTY SHEET_NSELECTED` keyword.

Using `WIDGET_INFO`, it is also possible to return the identifiers of all selected properties using the `/PROPERTY SHEET_SELECTED` keyword. This returns a string or string array containing the identifiers of the selected properties.

```

; Return information about single or multiple property
; selections.
vNumSelected = WIDGET_INFO(event.ID, /PROPERTY SHEET_NSELECTED)
vSelected = WIDGET_INFO(event.ID, /PROPERTY SHEET_SELECTED)
PRINT, 'Number properties selected: ' + STRING(vNumSelected)
PRINT, 'Selected properties: '
PRINT, vSelected

```

Controlling When Properties are Selectable

Three things that determine the appearance of a property sheet data cells. They are, in order of greatest to least precedence:

1. Sensitivity of the entire widget — If `SENSITIVE=0` for `WIDGET_PROPERTY SHEET` then no selection or scrolling is possible.
2. Editability of the entire widget — If `EDITABLE=0` for the property sheet (meaning it is marked as read-only), cells can be selected but cannot be changed. If `EDITABLE=1` (the default value meaning properties can be selected and modified), then the editability of individual properties is controlled by their individual sensitivity values.
3. Sensitivity of an individual property — If `SENSITIVE=0` for an individual property (set using the `RegisterProperty` or `SetPropertyAttribute` methods of `IDLitComponent`), then the individual property cannot be selected or changed.

Changing Properties

A change event is generated whenever a new value is entered for a property. It is also used to signal that a user-defined property needs changing. The event structure (`WIDGET_PROPSHEET_CHANGE`) provided when a change occurs contains a `COMPONENT`, an `IDENTIFIER`, a `PROPTYPE`, and a `SET_DEFINED` tag. The `COMPONENT` tag contains a reference to the object associated with the property sheet. When multiple objects are associated with the property sheet, this member indicates which object is to change. The `IDENTIFIER` tag specifies the value of the property's identifier attribute. This identifier is unique among all of the component's properties. The `PROPTYPE` tag indicates the type of the property (integer, string, etc.). Integer values for these types can be found in the documentation for components. The `SET_DEFINED` tag indicates whether or not an undefined property is having its value set. In most circumstances, along with its new value, the property should have its 'UNDEFINED' attribute set to zero. If a property is never marked as undefined, this field can be ignored.

Although the component's object reference is included in the event structure, it can also be retrieved via the following call:

```
WIDGET_CONTROL, event.id, GET_VALUE = obj
```

where `event` is the event structure and `obj` is the object reference of the component.

The `PROPTYPE` field is provided for convenience. The property type should be known implicitly based on `IDENTIFIER`, but can be retrieved (in integer form) by:

```
obj->GetPropertyAttribute, event.identifier, TYPE = type
```

where `obj` is the object reference of the component, `event` is the event structure, and `type` represents the data type of the property. Here, the value returned in by the `TYPE` keyword is the same as the value of the `PROPTYPE` field of the widget event structure.

Properties can use their `UNDEFINED` attribute to show an indeterminate state (set attribute `UNDEFINED = 1`). This might arise after the aggregation of two or more properties. One could imagine a `COLOR` property representing both the border and the interior color of a polygon so that just one color property is displayed in the property sheet. When set, the chosen color would be applied to both, and then the following code could be used to mark the property as defined:

```
IF (event.set_defined) THEN $
    event.component->SetPropertyAttribute, $
        event.identifier, UNDEFINED = 0
WIDGET_CONTROL, event.id, REFRESH_PROPERTY = event.identifier
```

where `event` is the event structure.

Note

The `REFRESH_PROPERTY` keyword to `WIDGET_CONTROL` is used to refresh the property sheet. This is necessary because although the property sheet knows about its component, it does not directly change the component itself. Just as with changing properties values, the property sheet and underlying component have a clear boundary and can only affect each other through IDL statements.

Properties can also be hidden (removing them from the property sheet entirely) or desensitized (displaying the property in the property sheet, but not allowing the user to change its value). See “[Property Attributes](#)” (Chapter 4, *iTool Developer’s Guide*) for additional details.

Updating the Component

When a value has been changed in the property sheet, you can access this resulting value through the `WIDGET_INFO` function:

```
value = WIDGET_INFO(event.id, PROPERTY_VALUE = event.identifier)
```

where `event` is the event structure. This value can then be used to update the changed property in the component object by calling its `SetPropertyByIdentifier` method:

```
event.component->SetPropertyByIdentifier, event.identifier, $
value
```

where `event` is the event structure and `value` is the modified property value.

User-defined Properties

User-defined properties allow IDL programmers to provide their own custom means for editing a property. One significant difference from other types of properties is that user-defined properties must have a string version of their value. This string value is stored in the `USERDEF` attribute of the property and must be explicitly updated. The string value is the value displayed in the property sheet. See [Chapter 4, “Property Management”](#) (*iTool Developer’s Guide*) for further discussion of user-defined properties.

Updating User-defined Properties

Like other property types, user-defined properties generate IDL property sheet change events. The difference is that the IDL event handler cannot query the property sheet for the new value. It must use some other means to determine a new value. Typically this is done through widget code, in which the user is asked to set a value, but virtually any other technique is valid.

When handling change events, determine the property’s type using the `PROPTYPE` field of the widget event structure. Once a value has been acquired, update the component using its `SetProperty` method. In addition, the string version of the user-defined property’s value should be updated. This is done by executing a statement similar to the following example:

```
eventBase.component->SetPropertyAttribute, $
    eventBase.identifier, USERDEF = userDefValue
```

where `eventBase` is the event structure of the top-level-base and `userDefValue` is the string representing the user-defined value when the property sheet is refreshed.

Once the underlying component has been updated, the property sheet is ready to be refreshed. Execute a call to update a given property with the current value:

```
WIDGET_CONTROL, propsheet, REFRESH_PROPERTY = eventBase.identifier
```

where `propsheet` is the widget ID of the property sheet widget and `eventBase` is the event structure of the top-level-base.

Property Sheet Sizing

Property sheets without a size definition (lacking a specified `SCR_XSIZE` or `XSIZE` keyword value) are naturally sized. Column widths are dependent upon the cell contents of the components. Naturally sized property sheets allow the full contents of the longest cell to be visible in a column as shown in the left-hand image in the following figure. When a size definition is provided, selecting the cell displays the list contents in a drop-down box that is wide enough for the longest item as shown in the right-hand image in the following figure.

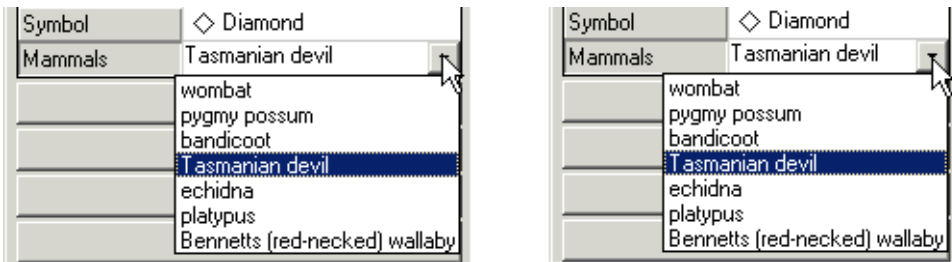


Figure 6-2: Property Sheet Column Sizing

Note

If you manually change the width of a property sheet column, natural resizing functionality is overridden. Dynamic resizing is not supported when the property sheet is refreshed or loaded with different data. Natural sizing can be recovered by destroying and recreating the property sheet.

The following elements are considered when determining column width in a naturally sized property sheet:

- The column width is dependent upon the length of the longest cell value (regardless of cell type) or longest component name. If the length is excessive, a reasonable default is used. When there are multiple components, only the data in the first three component columns are considered when determining column width. All columns will be the same width.
- If a text cell contains the longest value, approximately 25 characters will be displayed. When you click in the cell, a drop-down box shows any additional text. A scroll bar is provided to show text beyond that displayed in the drop-down box.

- If a drop-down list contains the longest value, the width of the longest enumerated value will determine the column width.
- If a number cell contains the longest value, ten digits plus the “.” and “-” characters will be displayed.
- If the longest cell value is in a cell containing color, symbol, line thickness, or line style items, then the column width is intelligently sized to allow the minimal width required for user identification and selection.

When a property sheet size is explicitly defined, the column width may crop the display of the full cell contents. However, when you select the cell, the full contents will be visible as follows:

- A drop-down list box will expand to show the longest item if the column width is less than the width of the longest item.
- A drop-down edit box will wrap text and provide a vertical scrollbar as necessary.

Property Sheet Example

The following example provides a property sheet containing all the available controls, including user-defined properties of a custom component.

Enter the following text into the IDL Editor:

```

; Property Sheet Demo
;
; This program contains these sections of code:
;
; (1) Definition of the IDLitTester class.
; (2) Methods for handling the user-defined data type.
; (3) Event handlers and main widget program.

;=====
; (1) Definition of the IDLitTester class.
;-----
; IDLitTester
;
; Superclasses:
;   IDLitComponent
;
; Subclasses:
;   none
;
; Interfaces:
;   IIDLProperty
;
; Intrinsic Methods:
; none (because it contains no objects)

;-----
; IDLitTester::Init

FUNCTION IDLitTester::Init, _REF_EXTRA = _extra

compile_opt idl2

; Initialize the superclass.
IF (self->IDLitComponent::Init() ne 1) THEN $
    RETURN, 0

; Create IDLitTester.
; Nothing to do, for now.

; Register properties.
;

```



```

; * Only registered properties will show up in the property sheet.
; * <identifier> must match self.<identifier>.

self->RegisterProperty, 'BOOLEAN', /BOOLEAN, $
    NAME = 'Boolean', DESCRIPTION = 'TRUE or FALSE'

self->RegisterProperty, 'COLOR', /COLOR, $
    NAME = 'Color', DESCRIPTION = 'Color (RGB)'

self->RegisterProperty, 'USERDEF', USERDEF = '', $
    NAME = 'User Defined', DESCRIPTION = 'User defined property'

self->RegisterProperty, 'NUMBER1', /INTEGER, $
    NAME = 'Integer', DESCRIPTION = 'Integer in [-100, 100]', $
    valid_range = [-100, 100]

self->RegisterProperty, 'NUMBER2', /FLOAT, $
    NAME = 'Floating Point', DESCRIPTION = 'Number trackbar', $
    valid_range = [-19.0D, 6.0D, 0.3333333333333333D]

self->RegisterProperty, 'NUMBER3', /FLOAT, $
    NAME = 'Floating Point', $
    DESCRIPTION = 'Double in [-1.0, 1.0]', $
    valid_range = [-1.0D, 1.0D]

self->RegisterProperty, 'LINESTYLE', /LINESTYLE, $
    NAME = 'Line Style', DESCRIPTION = 'Line style'

self->RegisterProperty, 'LINETHICKNESS', /THICKNESS, $
    NAME = 'Line Thickness', $
    DESCRIPTION = 'Line thickness (pixels)'

self->RegisterProperty, 'STRINGOLA', /STRING, $
    NAME = 'String', DESCRIPTION = 'Just some text'

self->RegisterProperty, 'SYMBOL', /SYMBOL, $
    NAME = 'Symbol', DESCRIPTION = 'Symbol of some sort'

self->RegisterProperty, 'STRINGLIST', $
    NAME = 'String List', DESCRIPTION = 'Enumerated list', $
    enumlist = ['dog', 'cat', 'bat', 'rat', 'nat', $
    'emu', 'owl', 'pig', 'hog', 'ant']

; Set any property values.
self->SetProperty, _EXTRA = _extra

RETURN, 1
END

```

```

;-----
; IDLitTester::Cleanup

PRO IDLitTester::Cleanup

compile_opt idl2

self->IDLitComponent::Cleanup

END

;-----
; IDLitTester::GetProperty
;
; Implementation for IIDLProperty interface

PRO IDLitTester::GetProperty, $
    boolean = boolean, $
    color = color, $
    userdef = userdef, $
    font = font, $
    number1 = number1, $
    number2 = number2, $
    number3 = number3, $
    linestyle = linestyle, $
    linethickness = linethickness, $
    stringola = stringola, $
    stringlist = stringlist, $
    symbol = symbol, $
    _REF_EXTRA = _extra

compile_opt idl2

IF (arg_present(boolean)) THEN boolean = self.boolean
IF (arg_present(color)) THEN color = self.color
IF (arg_present(userdef)) THEN userdef = self.userdef
IF (arg_present(font)) THEN font = self.font
IF (arg_present(number1)) THEN number1 = self.number1
IF (arg_present(number2)) THEN number2 = self.number2
IF (arg_present(number3)) THEN number3 = self.number3
IF (arg_present(linestyle)) THEN linestyle = self.linestyle
IF (arg_present(linethickness)) $
    THEN linethickness = self.linethickness
IF (arg_present(stringola)) THEN stringola = self.stringola
IF (arg_present(stringlist)) THEN stringlist = self.stringlist
IF (arg_present(symbol)) THEN symbol = self.symbol

; Superclass' properties:
IF (n_elements(_extra) gt 0) THEN $

```

```

        self->IDLitComponent::GetProperty, _EXTRA = _extra

END

;-----
; IDLitTester::SetProperty
;
; Implementation for IIDLProperty interface

PRO IDLitTester::SetProperty, $
    boolean = boolean, $
    color = color, $
    userdef = userdef, $
    font = font, $
    number1 = number1, $
    number2 = number2, $
    number3 = number3, $
    linestyle = linestyle, $
    linethickness = linethickness, $
    stringola = stringola, $
    stringlist = stringlist, $
    symbol = symbol, $
    _REF_EXTRA = _extra

compile_opt idl2

IF (n_elements(boolean) ne 0) THEN self.boolean = boolean
IF (n_elements(color) ne 0) THEN self.color = color
IF (n_elements(userdef) ne 0) THEN self.userdef = userdef
IF (n_elements(font) ne 0) THEN self.font = font
IF (n_elements(number1) ne 0) THEN self.number1 = number1
IF (n_elements(number2) ne 0) THEN self.number2 = number2
IF (n_elements(number3) ne 0) THEN self.number3 = number3
IF (n_elements(linestyle) ne 0) THEN self.linestyle = linestyle
IF (n_elements(linethickness) ne 0) THEN $
    self.linethickness = linethickness
IF (n_elements(stringola) ne 0) THEN self.stringola = stringola
IF (n_elements(stringlist) ne 0) THEN self.stringlist = stringlist
IF (n_elements(symbol) ne 0) THEN self.symbol = symbol

self->IDLitComponent::SetProperty, _EXTRA = _extra

END

;-----
; IDLitTester__Define

PRO IDLitTester__Define

```

```

compile_opt idl2, hidden

struct = {$
  IDLitTester, $
  inherits IDLitComponent, $
  boolean:0L, $
  color:[0B,0B,0B], $
  userdef:"", $
  number1:0L, $
  number2:0D, $
  number3:0D, $
  linestyle:0L, $
  linethickness:0L, $
  stringola:"", $
  stringlist:0L, $
  symbol:0L $
}

END

;=====
; (2) Methods for handling the user-defined data type.
;-----
; UserDefEvent
;
; This procedure is just part of the widget code for
; the user defined property.

PRO UserDefEvent, e

IF (tag_names(e, /structure_name) eq 'WIDGET_BUTTON') $
  THEN BEGIN

  widget_control, e.top, get_uvalue = uvalue
  widget_control, e.id, get_uvalue = numb_ness

  propsheet = uvalue.propsheet
  component = uvalue.component
  identifier = uvalue.identifier

  ; Set the human readable value.
  component->SetPropertyAttribute, $
    identifier, userdef = numb_ness

  ; Set the real value of the component.
  component->SetPropertyByIdentifier, identifier, numb_ness

  WIDGET_CONTROL, propsheet, refresh_property = identifier
  PRINT, 'Changed: ', uvalue.identifier, ': ', numb_ness

```

```

        WIDGET_CONTROL, e.top, /destroy

ENDIF

END

;-----
; GetUserDefValue
;
; Creates widgets used to modify the user defined property's
; value. The value is actually set in UserDefEvent.

PRO GetUserDefValue, e

base = WIDGET_BASE(/row, title = 'Pick a Number', $
        /modal, group_leader = e.top)

one = WIDGET_BUTTON(base, value = 'one', uvalue = 'oneness')
two = WIDGET_BUTTON(base, value = 'two', uvalue = 'twoness')
six = WIDGET_BUTTON(base, value = 'six', uvalue = 'sixness')
ten = WIDGET_BUTTON(base, value = 'ten', uvalue = 'tenness')

; We will need this info when we set the value
WIDGET_CONTROL, base, $
        SET_UVALUE = {propsheet:e.id, $
        component:e.component, $
        identifier:e.identifier}

WIDGET_CONTROL, base, /REALIZE

XMANAGER, 'UserDefEvent', base, event_handler = 'UserDefEvent'

END

;=====
; (3) Event handlers and main widget program.
;-----
;
; Event handling code for the main widget program and
; the main widget program.

;-----
; prop_event
;
; The property sheet generates an event whenever the user changes
; a value. The event holds the property's identifier and type, and
; an object reference to the component.
;
; Note: widget_control, e.id, get_value = objref also retrieves an

```

```

; object reference to the component.

PRO prop_event, e

IF (e.type eq 0) THEN BEGIN    ; Value changed

; Get the value of the property identified by e.identifier.

    IF (e.proptype ne 0) THEN BEGIN

        ; Get the value from the property sheet.
        value = widget_info(e.id, property_value = e.identifier)

        ; Set the component's property's value.
        e.component->SetPropertyByIdentifier, e.identifier, $
            value

        ; Print the change in the component's property value.
        PRINT, 'Changed', e.identifier, ': ', value
    ENDIF ELSE BEGIN

        ; Use alternative means to get the value.
        GetUserDefValue, e

    ENDELSE

ENDIF ELSE BEGIN                ; selection changed

    PRINT, 'Selected: ' + e.identifier
    r = e.component->GetPropertyByIdentifier(e.identifier, value)
    PRINT, ' Current Value: ', value

ENDELSE

END

;-----
; refresh_event

PRO refresh_event, e

WIDGET_CONTROL, e.id, get_uvalue = uvalue

uvalue.o->SetProperty, boolean = 0L
uvalue.o->SetProperty, color = [255, 0, 46]
uvalue.o->SetPropertyAttribute, 'userdef', userdef = "Yeehaw!"
uvalue.o->SetProperty, number1 = 99L
uvalue.o->SetProperty, number2 = -13.1
uvalue.o->SetProperty, number3 = 6.5

```

```

uvalue.o->SetProperty, linestyle = 6L
uvalue.o->SetProperty, stringola = 'It worked!'
uvalue.o->SetProperty, stringlist = 6L
uvalue.o->SetProperty, symbol = 6L

uvalue.o->SetPropertyAttribute, 'Number1', sensitive = 1
uvalue.o->SetPropertyAttribute, 'Number2', sensitive = 1

WIDGET_CONTROL, uvalue.prop, $
    REFRESH_PROPERTY = ['boolean', 'color', 'userdef', $
        'number1', 'number2', 'number3', 'linestyle', $
        'stringola', 'stringlist', 'symbol']

END

;-----
; reload_event

PRO reload_event, e

WIDGET_CONTROL, e.id, GET_UVALUE = uvalue

LoadValues, uvalue.o

WIDGET_CONTROL, uvalue.prop, SET_VALUE = uvalue.o

update_state, e.top, 1

END

;-----
; hide_event

PRO hide_event, e

WIDGET_CONTROL, e.id, get_uvalue = uvalue

uvalue.o->SetPropertyAttribute, 'color', /HIDE

WIDGET_CONTROL, uvalue.prop, refresh_property = 'color'

END

;-----
; show_event

PRO show_event, e

WIDGET_CONTROL, e.id, get_uvalue = uvalue

```

```

uvalue.o->SetPropertyAttribute, 'color', hide = 0

WIDGET_CONTROL, uvalue.prop, REFRESH_PROPERTY = 'color'

END

;-----
; clear_event

PRO clear_event, e

update_state, e.top, 0

WIDGET_CONTROL, e.id, GET_UVALUE = uvalue

WIDGET_CONTROL, uvalue.prop, SET_VALUE = OBJ_NEW()

END

;-----
; psdemo_large_event
;
; Handles resize events for the property sheet demo program.

PRO psdemo_large_event, e

WIDGET_CONTROL, e.id, GET_UVALUE = base
geo_tlb = WIDGET_INFO(e.id, /GEOMETRY)

WIDGET_CONTROL, base.prop, $
    SCR_XSIZE = geo_tlb.xsize - (2*geo_tlb.xpad), $
    SCR_YSIZE = geo_tlb.ysize - (2*geo_tlb.ypad)

END

;-----
; sensitivity_event
;
; Procedure to test sensitizing and desensitizing

PRO sensitivity_event, e

WIDGET_CONTROL, e.id, GET_UVALUE = uvalue, GET_VALUE = value

IF (value eq 'Desensitize') THEN b = 0 $
ELSE b = 1

uvalue.o->SetPropertyAttribute, 'Boolean', sensitive = b

```



```

uvalue.o->SetPropertyAttribute, 'Color', sensitive = b
uvalue.o->SetPropertyAttribute, 'UserDef', sensitive = b
uvalue.o->SetPropertyAttribute, 'Number1', sensitive = b
uvalue.o->SetPropertyAttribute, 'Number2', sensitive = b
uvalue.o->SetPropertyAttribute, 'Number3', sensitive = b
uvalue.o->SetPropertyAttribute, 'LineStyle', sensitive = b
uvalue.o->SetPropertyAttribute, 'LineThickness', sensitive = b
uvalue.o->SetPropertyAttribute, 'Stringola', sensitive = b
uvalue.o->SetPropertyAttribute, 'Symbol', sensitive = b
uvalue.o->SetPropertyAttribute, 'StringList', sensitive = b

WIDGET_CONTROL, uvalue.prop, $
    refresh_property = ['Boolean', 'Color', 'UserDef', $
        'Number1', 'Number2', 'Number3', 'LineStyle', $
        'LineThickness', 'Stringola', 'Symbol', 'StringList']

END

;-----
; LoadValues

PRO LoadValues, o

o->SetProperty, boolean = 1L          ; 0 or 1
o->SetProperty, color = [200, 100, 50] ; RGB
o->SetPropertyAttribute, 'userdef', userdef = ""
; to be set later
o->SetProperty, number1 = 42L         ; integer
o->SetProperty, number2 = 0.0        ; double
o->SetProperty, number3 = 0.1        ; double
o->SetProperty, linestyle = 4L       ; 5th item (zero based)
o->SetProperty, linethickness = 4L    ; pixels
o->SetProperty, stringola = "This is a silly string."
o->SetProperty, stringlist = 3L      ; 4th item in list
o->SetProperty, symbol = 4L          ; 5th symbol in list

END

;-----
; quit_event

PRO quit_event, e

WIDGET_CONTROL, e.top, /DESTROY

END

;-----
; update_state

```

```

PRO update_state, top, sensitive

WIDGET_CONTROL, top, GET_UVALUE = uvalue

FOR i = 0, n_elements(uvalue.b) - 1 do $
    WIDGET_CONTROL, uvalue.b[i], sensitive = sensitive

END

;-----
; psdemo_large

PRO psdemo_large

; Create and initialize the component.

o = OBJ_NEW('IDLitTester')

LoadValues, o

; Create some widgets.

base = WIDGET_BASE(/COLUMN, /TLB_SIZE_EVENT, $
    TITLE = 'Property Sheet Demo (Large)')

prop = WIDGET_PROPERTYSHEET(base, value = o, $
    YSIZE = 13, /FRAME, event_pro = 'prop_event')

b1 = WIDGET_BUTTON(base, value = 'Refresh', $
    uvalue = {o:o, prop:prop}, $
    event_pro = 'refresh_event')

b2 = WIDGET_BUTTON(base, value = 'Reload', $
    uvalue = {o:o, prop:prop}, $
    event_pro = 'reload_event')

b3 = WIDGET_BUTTON(base, value = 'Hide Color', $
    uvalue = {o:o, prop:prop}, $
    event_pro = 'hide_event')

b4 = WIDGET_BUTTON(base, value = 'Show Color', $
    uvalue = {o:o, prop:prop}, $
    event_pro = 'show_event')

b5 = WIDGET_BUTTON(base, value = 'Clear', $
    uvalue = {o:o, prop:prop}, $
    event_pro = 'clear_event')

```

```

b6 = WIDGET_BUTTON(base, value = 'Desensitize', $
    uvalue = {o:o, prop:prop}, $
    event_pro = 'sensitivity_event')

b7 = WIDGET_BUTTON(base, value = 'Sensitize', $
    uvalue = {o:o, prop:prop}, $
    event_pro = 'sensitivity_event')

b8 = WIDGET_BUTTON(base, value = 'Quit', $
    EVENT_PRO = 'quit_event')
; Buttons that can't be pushed after clearing:
b = [b1, b3, b4, b5, b6, b7]

; Activate the widgets.

WIDGET_CONTROL, base, SET_UVALUE = {prop:prop, b:b}, /REALIZE

XMANAGER, 'psdemo_large', base, /NO_BLOCK

END

```

The following figure displays the output of this example:

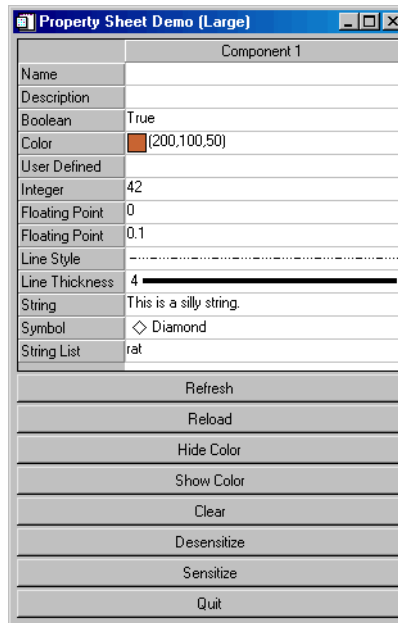


Figure 6-3: User-Defined Property Sheet Example

To demonstrate the controls available from the `WIDGET_PROPERTY SHEET`, do the following and note the `Selected` and `Changed` messages in the IDL Output Log:

- Change the **Boolean** field to **False**.
- Select a new **Color** from the color picker.
- Select a new “numberness” value in the **User Defined** field.
- Change the **Integer** field to a new value. Note that this field has been restricted to integers in the range -100 to 100.
- Change the first **Floating Point** field to a new value by moving the slider.
- Change the second **Floating Point** field to a new value by editing the text. Note that this field has been restricted to floating point numbers in the range -1.0 to 1.0.
- Change the **Line Style** field to a new style.
- Change the **Line Thickness** field to a new thickness.
- Select a new symbol in the **Symbol** field.
- Select a new string from the **String List**.

Click the eight buttons at the bottom of the property sheet to initiate the following events:

- The **Refresh** button loads the data specified in `refresh_event` into the property sheet, using the `REFRESH_PROPERTY` keyword to `WIDGET_CONTROL`.
- The **Reload** button reloads the data specified in `LoadValues` into the property sheet, using the `SET_VALUE` keyword to `WIDGET_CONTROL`.
- The **Hide Color** button runs `hide_event`, which sets the `HIDE` attribute for the color property to one.
- The **Show Color** button runs `show_event`, which sets the `HIDE` attribute for the color property to zero.
- The **Clear** button runs `clear_event`, which creates a new set of empty objects, deactivating all but the **Reload** button.
- The **Desensitize** button runs `sensitivity_event`, which deactivates the displayed fields.
- The **Sensitize** button runs `sensitivity_event`, which reactivates the displayed fields.

- The **Quit** button runs `quit_event`, which destroys the top-level base and ends the program.

Multiple Properties Example

The following example shows how to create a property sheet for multiple components.

Enter the following text in the IDL Editor:

```

; ExMultiSheet.pro
;
; Provides an example of a property sheet that is
; associated with more than one object. In this case,
; multiple IDLitVisAxis objects are used, with random
; colors and hidden cells, just for fun.

PRO PropertyEvent, event

IF (event.type EQ 0) THEN BEGIN      ; Value changed.

    PRINT, 'Changed: ', event.component
    PRINT, '    ', event.identifier, ': ', $
    WIDGET_INFO(event.id, COMPONENT = event.component, $
        PROPERTY_VALUE = event.identifier)

ENDIF ELSE BEGIN                    ; Selection changed.

    PRINT, 'Selected: ' + event.identifier

ENDELSE

END

PRO CleanupEvent, baseID

WIDGET_CONTROL, baseID, GET_UVALUE = objects

FOR i = 0, (N_ELEMENTS(objects) - 1) DO $
    OBJ_DESTROY, objects[i]

END

PRO ExMultiSheet_event, event

ps = WIDGET_INFO(event.id, $
    FIND_BY_UNAME = 'PropSheet')

geo_tlb = WIDGET_INFO(event.id, /GEOMETRY)

WIDGET_CONTROL, ps, $

```

```

SCR_XSIZE = geo_tlb.xsize - (2*geo_tlb.xpad), $
SCR_YSIZE = geo_tlb.yysize - (2*geo_tlb.yypad)

END

PRO ExMultiSheet

tlb = WIDGET_BASE(/COLUMN, /TLB_SIZE_EVENTS, $
    KILL_NOTIFY = 'CleanupEvent')

; Create some columns.

oComp1 = OBJ_NEW('IDLitVisAxis', $
    COLOR = RANDOMU(s1, 3)*256, $
    TEXT_COLOR = RANDOMU(s7, 3)*256)
oComp2 = OBJ_NEW('IDLitVisAxis', $
    COLOR = RANDOMU(s2, 3)*256, $
    TEXT_COLOR = RANDOMU(s8, 3)*256)
oComp3 = OBJ_NEW('IDLitVisAxis', $
    COLOR = RANDOMU(s3, 3)*256, $
    TEXT_COLOR = RANDOMU(s9, 3)*256)
oComp4 = OBJ_NEW('IDLitVisAxis', $
    COLOR = RANDOMU(s4, 3)*256, $
    TEXT_COLOR = RANDOMU(s10, 3)*256)
oComp5 = OBJ_NEW('IDLitVisAxis', $
    COLOR = RANDOMU(s5, 3)*256, $
    TEXT_COLOR = RANDOMU(s11, 3)*256)
oComp6 = OBJ_NEW('IDLitVisAxis', $
    COLOR = RANDOMU(s6, 3)*256, $
    TEXT_COLOR = RANDOMU(s12, 3)*256)

oComps = [oComp1, oComp2, oComp3, $
    oComp4, oComp5, oComp6]

WIDGET_CONTROL, tlb, SET_UVALUE = oComps

; Hide some properties.

oComp2->SetPropertyAttribute, 'color', /HIDE
oComp2->SetPropertyAttribute, 'ticklen', /HIDE
oComp5->SetPropertyAttribute, 'ticklen', /HIDE

; Create the property sheet.
prop = WIDGET_PROPERTY SHEET(tlb, $
    UNAME = 'PropSheet', $
    VALUE = oComps, $
    FONT = 'Courier New*16', $
    XSIZE = 100, YSIZE = 24, $
    /FRAME, EVENT_PRO = 'PropertyEvent')

```

```

; Activate the widgets.

WIDGET_CONTROL, tlb, /REALIZE

XMANAGER, 'ExMultiSheet', tlb, /NO_BLOCK

END

```

Save the program as `ExMultiSheet.pro`, then compile and run it. A property sheet displaying the properties of six axes is displayed:

	Axis	Axis	Axis	Axis	Axis	Axis
Name	Axis	Axis	Axis	Axis	Axis	Axis
Description	Axis Visualiza	Axis Visualiza	Axis Visualiza	Axis Visualiza	Axis Visualiza	Axis Visualiza
Hide	Show	Show	Show	Show	Show	Show
Major tick length	0.05		0.05	0.05		0.05
Title						
Text color	(136, 55, 108)	(186, 3, 54)	(74, 143, 255)	(229, 136, 196)	(10, 222, 91)	(207, 174, 211)
Axis color	(228, 55, 220)		(49, 95, 165)	(175, 247, 216)	(134, 3, 52)	(230, 232, 212)
Color palette						
Line style						
Line thickness	1	1	1	1	1	1
Use logarithmic axis	False	False	False	False	False	False
Use exact axis range	True	True	True	True	True	True
Extend axis	False	False	False	False	False	False
Number of major ticks	6	6	6	6	6	6
Number of minor ticks	3	3	3	3	3	3
Minor tick length	0.5	0.5	0.5	0.5	0.5	0.5
Tick interval	0	0	0	0	0	0
Tick layout	Axis plus	Axis plus	Axis plus	Axis plus	Axis plus	Axis plus
Tick format code						
Text hide	False	False	False	False	False	False
Text position	Below/left	Below/left	Below/left	Below/left	Below/left	Below/left
Text font	Helvetica	Helvetica	Helvetica	Helvetica	Helvetica	Helvetica
Text style	Normal	Normal	Normal	Normal	Normal	Normal
Text font size	12	12	12	12	12	12

Figure 6-4: Multi-Sheet Example

The gray boxes indicate properties that have been hidden. To remove the gray boxes, comment out the code after the following comment:

```

; Hide some properties.

```

The text is displayed at 16 points in the Courier New font. To view the property sheet with the text displayed in the default size and font, comment out the following segment of the property sheet creation code:

```

FONT = "Courier New*16", $

```

To see the text displayed in a font and size of your choosing, edit the same segment to include a different font name and size.



Chapter 7

Using Table Widgets

The following topics are covered in this chapter:

Using Table Widgets	154	Edit Mode	162
Default Table Size	155	Cell Attributes	163
Selection Modes	156	Example: Single Data Type Data	170
Data Types	158	Example: Structure Data	174
Data Retrieval	159		

Using Table Widgets

Table widgets display two-dimensional data and allow in-place data editing.

See “[WIDGET_TABLE](#)” (*IDL Reference Guide*) for a complete description of the function used to create table widgets.

This section discusses the following topics:

- “[Default Table Size](#)” on page 155
- “[Selection Modes](#)” on page 156
- “[Data Types](#)” on page 158
- “[Data Retrieval](#)” on page 159
- “[Edit Mode](#)” on page 162
- “[Cell Attributes](#)” on page 163
- “[Example: Single Data Type Data](#)” on page 170
- “[Example: Structure Data](#)” on page 174

Default Table Size

Table widgets are sized according to the value of the following pairs of keywords to `WIDGET_TABLE`, in order of precedence: `SCR_XSIZE/SCR_YSIZE`, `XSIZE/YSIZE`, `X_SCROLL_SIZE/Y_SCROLL_SIZE`, and `VALUE`. If either dimension remains unspecified by one of the above keywords, the default value of six (columns or rows) is used when the table is created. If the width or height specified is less than the size of the table, scroll bars are added automatically.

Note

The default row height and column width vary with different user interface toolkits.

Selection Modes

Groups of table cells can be selected either manually (using the mouse or keyboard) or programmatically. The table widget supports two selection modes — *standard* and *disjoint*. Both modes can be used either by an interactive table user or by the IDL programmer. See “[Data Retrieval](#)” on page 159 for information on retrieving data from various types of selections.

Standard Selection Mode

In standard selection mode, exactly one rectangular area (of a single cell or multiple cells) can be selected at a given time.

Interactive Selection

Interactive users select cells by clicking the left mouse button on a cell, holding the mouse button down, and dragging the mouse until the desired cells are selected. Selections can be extended by holding down the SHIFT key and selecting additional cells.

Programmatic Selection

Programmers select cells by specifying a four-element array, of the form [*left, top, right, bottom*], as the value of the [SET_TABLE_SELECT](#) keyword to WIDGET_CONTROL.

Disjoint Selection Mode

In disjoint selection mode, multiple rectangular areas can be selected at once. In order to place a table in disjoint selection mode, the programmer must either specify the [DISJOINT_SELECTION](#) keyword to WIDGET_TABLE when creating the table, or set the [TABLE_DISJOINT_SELECTION](#) keyword to WIDGET_CONTROL after the table has been created.

Interactive Selection

Interactive users select multiple disjoint cell regions by:

1. Creating an initial selection as described above.
2. Holding down the CONTROL key and selecting an unselected cell by clicking and holding down the left mouse button.

3. Releasing the CONTROL key (while continuing to hold the mouse button down) and dragging the mouse until the next desired region is selected.
4. Repeating as necessary.

Selections can be extended by holding down the SHIFT key and selecting additional cells.

Programmatic Selection

Programmers create select multiple disjoint cell regions by providing a $2 \times n$ element array of column/row pairs specifying the cells to act upon as the value of the `SET_TABLE_SELECT` keyword to `WIDGET_CONTROL`.

Data Types

Table data can be of any IDL data type or types.

Single Data Type

If all of the table data is of the same data type, the table value is specified as a two-dimensional array.

Values returned by the `GET_VALUE` keyword to `WIDGET_CONTROL` are either a two-dimensional array (for full tables or selections when the table is in standard selection mode) or a one-dimensional array (for tables in disjoint selection mode). (See [“Data Retrieval”](#) on page 159 for details.)

Multiple Data Types

If the table contains data of several data types, the table value is specified as a vector of structures. All of the structures must be of the same type, and must contain one field for each row (if the `COLUMN_MAJOR` keyword to `WIDGET_TABLE` is set) or column (if the `ROW_MAJOR` keyword to `WIDGET_TABLE` is set; this is the default) in the table.

Values returned by the `GET_VALUE` keyword to `WIDGET_CONTROL` are either a vector of structures (for full tables or selections when the table is in standard selection mode) or a single structure with one field per cell (for selections when the table is in disjoint selection mode). (See [“Data Retrieval”](#) on page 159 for details.)

Data Retrieval

To retrieve data from a table widget, use the `GET_VALUE` keyword to `WIDGET_CONTROL`. You can retrieve the entire contents of the table or the contents of either a standard or disjoint selection. The format of the variable returned by the `GET_VALUE` keyword depends on the type of data displayed in the table (see “Data Types” on page 158) and the type of selection (see “Selection Modes” on page 156).

Entire Table

To retrieve data from the entire table, use the following command:

```
WIDGET_CONTROL, table, GET_VALUE=table_value
```

where *table* is the widget ID of the table widget. The *table_value* variable will contain either:

- an array with the same dimensions as the table, with one element per table cell, if the table contains data of a single data type, or
- a vector of structures, with one structure per table row or column, if the table contains structure data.

Standard Selection

To retrieve data for a group of selected cells, use the following command:

```
WIDGET_CONTROL, table, GET_VALUE=selection_value /USE_TABLE_SELECT
```

where *table* is the widget ID of the table widget. In standard selection mode, the *selection_value* variable will contain either:

- an array with the same dimensions as the selection, with one element per selected cell, if the table contains data of a single data type, or
- a vector of structures, with one structure per selected row or column, if the table contains structure data.

Note

You can also set the `USE_TABLE_SELECT` keyword equal to a four-element array of the form [*left*, *top*, *right*, *bottom*] containing the zero-based indices of the columns and rows that should be selected.

To retrieve the list of selected cells, use the following command:

```
selected_cells = WIDGET_INFO(table, /TABLE_SELECT)
```

where *table* is the widget ID of the table widget. The *selected_cells* variable will contain a four-element array of the form [*left*, *top*, *right*, *bottom*] containing the zero-based indices of the columns and rows that are selected.

Disjoint Selection

To retrieve data for a group of selected cells, use the following command:

```
WIDGET_CONTROL, table, GET_VALUE=selection_value,  
/USE_TABLE_SELECT
```

where *table* is the widget ID of the table widget. In disjoint selection mode, the *selection_value* variable will contain either:

- a one-dimensional array of values, with one element per selected cell, if the table contains data of a single data type, or
- a structure, with one field per selected cell, if the table contains structure data.

Note

You can also set the USE_TABLE_SELECT keyword equal to a 2 x *n* element array of column/row pairs specifying the cells that should be selected.

To retrieve the list of selected cells, use the following command:

```
selected_cells = WIDGET_INFO(table, /TABLE_SELECT)
```

where *table* is the widget ID of the table widget. The *selected_cells* variable will contain 2 x *n* array of column/row pairs containing the zero-based indices of the selected cells.

Converting Between Cell List Formats

With the addition of the ability to create disjoint table selections in IDL 5.6, the format of the list of selected cells returned by WIDGET_INFO was altered to accommodate non-rectangular regions when disjoint selections are enabled. To preserve backwards-compatibility, the format of the list was not changed for tables using standard selection mode, which guarantees a rectangular selection region.

If your application allows the table widget to switch between standard and disjoint selection mode, or if you have selection-handling routines that can be used with tables in either mode, you may want to modify the rectangular selection values returned for standard selections to match the lists of cells returned for disjoint

selections. The following is a template for such a utility function. It accepts a four-element array of the form [*left, top, right, bottom*] containing the zero-based indices of the columns and rows that are selected and converts it into a 2 x *n* array of column/row pairs containing the zero-based indices the selected cells.

```
FUNCTION Make_Cell_List, Selection_Vector
    num_cells = (Selection_Vector[2]-(Selection_Vector[0]-1)) * $
                (Selection_Vector[3]-(Selection_Vector[1]-1))
    return_arr = intarr(2,num_cells)
    n=0
    FOR i=Selection_Vector[1], Selection_Vector[3] DO BEGIN
        FOR j=Selection_Vector[0], Selection_Vector[2] DO BEGIN
            return_arr(n)=j
            return_arr(n+1)=i
            n=n+2
        ENDFOR
    ENDFOR
    RETURN, return_arr
END
```

With this function compiled, you could retrieve the four-element selection array from a standard selection and turn it into a 2 x *n* element array with the following commands:

```
selected_cells = WIDGET_INFO(table, /TABLE_SELECT)
cell_list = Make_Cell_List(selected_cells)
```

where *table* is the widget ID of a table widget in standard selection mode.

To reform the array returned by

```
WIDGET_CONTROL, table, GET_VALUE=Selection_Value
```

for a standard selection into one-dimensional array like those returned for disjoint selections, use the following command:

```
REFORM(Selection_Value, N_ELEMENTS(Selection_Value), 1)
```

Edit Mode

Edit mode allows a user to select and change the contents of a table cell. There are numerous ways to enter and exit Edit mode, including:

- Clicking on an unselected cell, then typing any character. This replaces the existing text with the new character.
- Clicking on an unselected cell, then typing a carriage return. This selects the contents of the cell and positions the cursor at the right. A second carriage return exits edit mode, making no changes.
- Double-clicking on an unselected cell. This selects the contents of the cell and positions the cursor at the right.
- Clicking on a selected cell. This selects the contents of the cell and positions the cursor at the right.
- Double-clicking on a selected cell. This positions the cursor at the position where the mouse pointer was clicked.

Cell Attributes

The table widget supports a variety of cell attributes that you can apply either to the entire table or to a subset of its cells. You can set them at table creation with the `WIDGET_TABLE` function and change them after creation with the `WIDGET_CONTROL` procedure. You can also query for some of them with the `WIDGET_INFO` function. The following table describes the table cell attributes.

Attribute	Description
ALIGNMENT	Horizontal alignment of text within a cell (left, middle, right)
BACKGROUND_COLOR	Color of the background of a cell
EDITABLE	Indication of whether you can edit a cell
FONT	Font to use when drawing a cell's text
FOREGROUND_COLOR	Color of the foreground of a cell
FORMAT	Formatting string to use when drawing a cell's value

Table 7-1: Table Cell Attributes

The following table indicates whether you can use these attributes with the widget creation (`WIDGET_TABLE`), modification (`WIDGET_CONTROL`), and querying (`WIDGET_INFO`) routines.

Attribute	WIDGET_TABLE	WIDGET_CONTROL	WIDGET_INFO
ALIGNMENT	Yes	Yes	No
BACKGROUND_COLOR	Yes	Yes	Yes
EDITABLE	Yes	Yes	Yes
FONT	Yes	Yes	Yes
FOREGROUND_COLOR	Yes	Yes	Yes
FORMAT	Yes	Yes	No

Table 7-2: Use of Table Cell Attributes with Key Widget Routines

There are a few issues surrounding table widget attributes that IDL programmers should be aware of, especially on Motif (UNIX) platforms. While it is expected that most users will not see any performance problems, you should consider the hardware limitations of your users' systems.

One issue is that the more cells a table has, the more sluggish the table can be. You can mitigate this limitation by operating on as few cells as possible. For example, if you know that all cells have the same background color, `WIDGET_INFO` need only query for the background color of one cell.

Another issue involves color on Motif systems. Depending on the graphics system in use, there might be only a small number of distinct colors available for the table.

Setting Cell Attributes at Table Creation

You can set table cell attributes when you create the widget. The value can be either a scalar or an array of values. The application and default use of cell attributes depends on which type you choose in specifying the value. (For descriptive purposes here, consider a color value to be a scalar, although in reality it is a three-element vector of bytes.) Here are the two scenarios:

- **The value is specified as a scalar** — applied to all cells. Additionally, it becomes the table's default value, used when new cells are created.
- **The value is specified as an array** — applied left to right, moving from the top row to the bottom row. The input array's dimensions need not match the table's dimensions. If the number of attribute value elements is insufficient for the number of cells, the values are recycled to apply to the cells, starting with the first value. If you supply more attribute values than there are cells, IDL does not use the remainder.

The following example shows how you can initialize a table to have the vintage look of alternating mint-green and white lines.

```
PRO minty_fresh

    t1b = WIDGET_BASE()

    rows = 5
    cols = 5

    ; Create 2-D array of background colors (3-D, actually)
    backgroundColors = MAKE_ARRAY( 3, cols, 2, /BYTE )

    backgroundColors[0,*,0] = 153    ; mint-green
    backgroundColors[1,*,0] = 255
```

```

backgroundColors[2,*,0] = 204

backgroundColors[*,*,1] = 255 ; white

; Create a table where every other line is mint-green
table = WIDGET_TABLE( tlb, $
    BACKGROUND_COLOR = backgroundColors, $
    VALUE = INDGEN(cols,rows) )

; Realize the widgets
WIDGET_CONTROL, tlb, /REALIZE

END

```

Note that in the example, only enough colors for the first two rows are specified. The table widget repeats the pattern for the remaining rows. Setting up a table with alternating column colors is even easier. To do so, you can create a table widget with the following line:

```
background_color = [ [153,255,204], [255,255,255] ]
```

If the example used this code, the table would have a checkerboard pattern because there are an odd number of columns.

The various types of cell attributes will try to convert input to the proper data type. The following table details the types and values that you should supply.

Attribute	Preferred Data Type	Value or Value Range
ALIGNMENT	BYTE	0, 1, or 2
BACKGROUND_COLOR	BYTE	RGB triplet whose elements are in the range of [0,255]
EDITABLE	BYTE	0 or non-zero
FONT	STRING	See “About Device Fonts” (Appendix H, <i>IDL Reference Guide</i>)
FOREGROUND_COLOR	BYTE	RGB triplet whose elements are in the range of [0,255]
FORMAT	STRING	See the FORMAT keyword of the PRINT procedure

Table 7-3: Preferred Data Types and Values of Table Cell Attributes

Users of table widgets should be aware of the following issues regarding cell attributes:

- When a cell value is edited, the foreground and background colors revert to the system defaults for edit cells. The font remains the same.
- At table creation, IDL automatically determines an optimal row height based on the fonts specified. However, an explicitly specified row height takes precedence. After creation, row heights do not automatically change when fonts change.
- Row and column header cells have a limited set of attributes. Only foreground color and font can be set. Header cells are indexed by using -1. For example, the header for the third row is indexed by [-1,2], and the third column is indexed by [2,-1]. Actions on cell [-1,-1] are ignored.
- On Windows, the number of distinct fonts is limited to 100. On UNIX systems, the table widget limits the number of distinct fonts to 1000 (although the user's particular system might have limitations).

Changing Cell Attributes after Table Creation

After you create a table widget, you can change cell attributes with the `WIDGET_CONTROL` procedure. There are two attribute-changing scenarios to consider:

- **Attributes are applied to a rectangular region of cells** — occurs when all of the table's cells are operated on because the `USE_TABLE_SELECT` keyword is not used, or occurs when that keyword is used and the table is in non-disjoint selection mode.

Regardless, an attribute keyword's value can be a scalar or an array and is applied repeatedly to the targeted cells. If the array of values is exhausted, it is recycled by wrapping back to the first value. Excess values are not used. The values are applied to the cells row by row (i.e., each row is completed before the next row begins). The ordering is left to right and top to bottom.

- **Attributes are applied to a list of cells** — occurs when `USE_TABLE_SELECT` is used and the table is in disjoint selection mode. As in the previous scenario, an attribute keyword's value can be a scalar or an array. In both cases, values are applied to cells by synchronously stepping through the list of cells and values. If the number of values is insufficient, IDL recycles them. IDL does not use excess values.

Note that when you specify scalar values, they do not replace the table's defaults. You can set a table cell attribute's defaults with the `WIDGET_TABLE` function. These

defaults come into play when you use the `INSERT_ROWS` and `INSERT_COLUMNS` keywords for `WIDGET_CONTROL`. New cells will have the defaults unless you also supply attribute keywords to override them.

If you change fonts, the table does not adjust row or column sizes to better fit the new font. You can programmatically change cell sizes with the `COLUMN_WIDTHS` and `ROW_HEIGHTS` keywords.

Note

The `ROW_HEIGHTS` keyword works on Windows with the limitation that all rows receive the same height.

You can also change the column header's height by specifying a value of -1 as the index for a cell's row. Unlike value cells, the column header can have a height that is different from the value-cell heights. You can control the row header's width by specifying -1 as the index for a cell's column.

The following example code demonstrates how you can modify a table's cell attributes. The example creates the table with all cells being editable, but changes that value so a rectangular region of six cells becomes uneditable (and grayed out to indicate that).

```
PRO alternate_editability

    tlb = WIDGET_BASE()

    ; Create a table
    table = WIDGET_TABLE( tlb, $
        /EDITABLE, $           ; all cells are editable
        VALUE = INDGEN(5,5) )

    ; Change the widget
    WIDGET_CONTROL, table, $
        EDITABLE=0, $           ; not editable
        BACKGROUND_COLOR=[223,223,223], $ ; gray them out
        USE_TABLE_SELECT=[1,1,3,2]      ; block of six cells

    ; Realize the widgets
    WIDGET_CONTROL, tlb, /REALIZE

end
```

IDL repeatedly applies the editability value and background colors to all six target cells. If the table were in disjoint selection mode, the `USE_TABLE_SELECT` line would look like this:

```
USE_TABLE_SELECT=[ [1,1], [2,1], [3,1], $
```

[1,2], [2,2], [3,2]]

Finally, if you want to apply the change to the current selection, /USE_TABLE_SELECT is sufficient.

Querying for Cell Attributes

You can retrieve certain table cell attribute values, thereby eliminating the need for the IDL programmer to independently keep track of these values. The `WIDGET_INFO` function can return information on the following cell attributes:

- `BACKGROUND_COLOR`
- `EDITABLE`
- `FONT`
- `FOREGROUND_COLOR`

The corresponding `WIDGET_INFO` keywords are preceded by “TABLE_” (e.g., `TABLE_BACKGROUND_COLOR`). You can query for only one attribute at a time. However, you can use the `USE_TABLE_SELECT` keyword conjunction with these attribute keywords to specify a set of cells to query. The following table shows how the dimensions of the returned variable depend on the table’s selection mode and the requested attribute.

Attribute Keyword	Table in Non-Disjoint Selection Mode; Selection Has M Columns and N Rows	Table in Disjoint Selection Mode; List of Selected Cells Has N Elements
<code>TABLE_BACKGROUND_COLOR</code>	3D array of bytes ($3 \times M \times N$)	2D array of bytes ($3 \times N$)
<code>TABLE_EDITABLE</code>	2D array of bytes ($M \times N$)	1D array of bytes (N)
<code>TABLE_FONT</code>	2D array of strings ($M \times N$)	1D array of strings (N)
<code>TABLE_FOREGROUND_COLOR</code>	3D array of bytes ($3 \times M \times N$)	2D array of bytes ($3 \times N$)

Table 7-4: Dimensions of WIDGET_INFO’s Return Variable with Use of USE_TABLE_SELECT and the Table Attribute Keywords

There is also a special case. If all values in the returned variable would be identical, `USE_TABLE_SELECT` returns a scalar (or a three-element array in the case of colors). This conglomeration preserves backward compatibility if you have used `WIDGET_INFO` to get a table's editability. If cell editability is not uniform for the queried cells, `WIDGET_INFO` returns an array, and conditionals using the returned value could throw an error. Therefore, if you make use of individual cell editability, you should check and test your code for these possible errors.

Similar to the way attribute values are applied to cells, `WIDGET_INFO` fills the return variable in a row-by-row fashion or with a one-to-one correspondence if the table is in disjoint selection mode.

Finally, for those cells that have not been explicitly assigned a value for the queried attribute, `WIDGET_INFO` returns the default value. This value is the one specified by `WIDGET_TABLE` or, failing that, the system default.

Adding and Deleting Cells

You can add and delete table cells through a variety of methods. One is to use the `WIDGET_CONTROL` procedure's row and column insertion and deletion keywords. Another is to change the table's `XSIZE` or `YSIZE` attributes. You can also make the change by setting the table's value. Regardless of the method, cell attributes shift appropriately with the change. Explicitly set row and column labels also shift, as do current column widths and row heights.

For example, suppose you insert a row above the current third row of an existing table by using the following statement:

```
WIDGET_CONTROL, myTable, $
  INSERT_ROWS = 1, $
  USE_TABLE_SELECT = [0,2,0,2]
```

After IDL executes this statement, the attributes of the first two rows are unchanged, and all of the other pre-existing rows have their attributes shifted down with them. Cells in the new row receive the table's default cell attributes.

Example: Single Data Type Data

The following procedures build a simple application that allows the user to select data from a table, plotting the data in a draw window and optionally displaying the data values in a text widget. The user can switch the table between standard and disjoint selection modes.

Example Code

This example is included in the file `table_widget_example1.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution. Run this example procedure by entering `table_widget_example1` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT table_widget_example1.pro`. See “[Running the Example Code](#)” on page 15 if IDL does not run the program as expected.

```

; Event-handler routine
PRO table_widget_example1_event, ev

    ; Retrieve the anonymous structure contained in the user value of
    ; the top-level base widget.
    WIDGET_CONTROL, ev.top, GET_UVALUE=stash

    ; Retrieve the table's selection mode and selection.
    disjoint = WIDGET_INFO(stash.table, /TABLE_DISJOINT_SELECTION)
    selection = WIDGET_INFO(stash.table, /TABLE_SELECT)

    ; Check to see whether a selection exists, setting the
    ; variable 'hasSelection' accordingly.
    IF (selection[0] ne -1) THEN hasSelection = 1 $
        ELSE hasSelection = 0

    ; If there is a selection, get the value.
    IF (hasSelection) THEN WIDGET_CONTROL, stash.table, $
        GET_VALUE=value, /USE_TABLE_SELECT

    ; The following sections define the application's reactions to
    ; various types of events.

    ; If the event came from the table, plot the selected data.
    IF ((ev.ID eq stash.table) AND hasSelection) THEN BEGIN
        WSET, stash.draw
        PLOT, value
    ENDIF

    ; If the event came from the 'Show Selected Data' button, display
    ; the data in the text widget.

```

```

IF ((ev.ID eq stash.b_value) AND hasSelection) THEN BEGIN
  IF (disjoint eq 0) THEN BEGIN
    WIDGET_CONTROL, stash.text, SET_VALUE=STRING(value, /PRINT)
  ENDIF ELSE BEGIN
    WIDGET_CONTROL, stash.text, SET_VALUE=STRING(value)
  ENDELSE
ENDIF

; If the event came from the 'Show Selected Cells' button,
; display the selection information in the text widget. Use
; different displays for standard and disjoint selections.
IF ((ev.ID eq stash.b_select) AND hasSelection) THEN BEGIN
  IF (disjoint eq 0) THEN BEGIN
    ; Create a string array containing the column and row
    ; values of the selected rectangle.
    list0 = 'Standard Selection'
    list1 = 'Left:   ' + STRING(selection[0])
    list2 = 'Top:    ' + STRING(selection[1])
    list3 = 'Right:  ' + STRING(selection[2])
    list4 = 'Bottom: ' + STRING(selection[3])
    list = [list0, list1, list2, list3, list4]
  ENDIF ELSE BEGIN
    ; Create a string array containing the column and row
    ; information for the selected cells.
    n = N_ELEMENTS(selection)
    list = STRARR(n/2+1)
    list[0] = 'Disjoint Selection'
    FOR j=0,n-1,2 DO BEGIN
      list[j/2+1] = 'Column: ' + STRING(selection[j]) + $
        ', Row: ' + STRING(selection[j+1])
    ENDFOR
  ENDELSE
  WIDGET_CONTROL, stash.text, SET_VALUE=list
ENDIF

; If the event came from the 'Change Selection Mode' button,
; change the table selection mode and the title of the button.
IF (ev.ID eq stash.b_change) THEN BEGIN
  IF (disjoint eq 0) THEN BEGIN
    WIDGET_CONTROL, stash.table, TABLE_DISJOINT_SELECTION=1
    WIDGET_CONTROL, stash.b_change, $
      SET_VALUE='Change to Standard Selection Mode'
  ENDIF ELSE BEGIN
    WIDGET_CONTROL, stash.table, TABLE_DISJOINT_SELECTION=0
    WIDGET_CONTROL, stash.b_change, $
      SET_VALUE='Change to Disjoint Selection Mode'
  ENDELSE
ENDIF

```

```

; If the event came from the 'Quit' button, close the
; application.
IF (ev.ID eq stash.b_quit) THEN WIDGET_CONTROL, ev.TOP, /DESTROY

END

; Widget creation routine.
PRO table_widget_example1

; Create data to be displayed in the table.
data = DIST(7)

; Create initial text to be displayed in the text widget.
help = ['Select data from the table below using the mouse.']

; Create the widget hierarchy.
base = WIDGET_BASE(/COLUMN)
subbase1 = WIDGET_BASE(base, /ROW)
draw = WIDGET_DRAW(subbase1, XSIZE=250, YSIZE=250)
subbase2 = WIDGET_BASE(subbase1, /COLUMN)
text = WIDGET_text(subbase2, XS=50, YS=8, VALUE=help, /SCROLL)
b_value = WIDGET_BUTTON(subbase2, VALUE='Show Selected Data')
b_select = WIDGET_BUTTON(subbase2, VALUE='Show Selected Cells')
b_change = WIDGET_BUTTON(subbase2, $
    VALUE='Change to Disjoint Selection Mode')
b_quit = WIDGET_BUTTON(subbase2, VALUE='Quit')
table = WIDGET_TABLE(base, VALUE=data, /ALL_EVENTS)

; Realize the widgets.
WIDGET_CONTROL, base, /REALIZE

; Get the widget ID of the draw widget.
WIDGET_CONTROL, draw, GET_VALUE=drawID

; Create an anonymous structure to hold widget IDs. This
; structure becomes the user value of the top-level base
; widget.
stash = {draw:drawID, table:table, text:text, b_value:b_value, $
    b_select:b_select, b_change:b_change, b_quit:b_quit}

; Set the user value of the top-level base and call XMANAGER
; to manage everything.
WIDGET_CONTROL, base, SET_UVALUE=stash
XMANAGER, 'table_widget_example1', base

END

```

The following things about this example are worth noting:

- It is important to check whether a selection exists before using `WIDGET_CONTROL` to retrieve the selection.
- Data from disjoint selections is handled differently than data from standard selections.
- For a relatively simple application, passing a group of widget IDs via the top-level base widget's user value allows the use of a single event routine rather than separate event routines for each widget.

Example: Structure Data

The following procedures build a simple application that displays the same structure data in two table widgets; one in row-major format and one in column-major format.

Example Code

This example is included in the file `table_widget_example2.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution. Run this example procedure by entering `table_widget_example2` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT table_widget_example2.pro`. See [“Running the Example Code”](#) on page 15 if IDL does not run the program as expected.

```

; Event-handler routine for 'Quit' button
PRO table_widget_example2_quit_event, ev
    WIDGET_CONTROL, ev.TOP, /DESTROY
END

; Widget creation routine.
PRO table_widget_example2

    ; Create some structure data.
    d0={planet:'Mercury', orbit:0.387, radius:2439, moons:0}
    d1={planet:'Venus', orbit:0.723, radius:6052, moons:0}
    d2={planet:'Earth', orbit:1.0, radius:6378, moons:1}
    d3={planet:'Mars', orbit:1.524, radius:3397, moons:2}

    ; Combine structure data into a vector of structures.
    data = [d0, d1, d2, d3]

    ; Create labels for the rows or columns of the table.
    labels = ['Planet', 'Orbit Radius (AU)', 'Radius (km)', 'Moons']

    ; To make sure the table looks nice on all platforms,
    ; set all column widths to the width of the longest string
    ; that can be a header.
    max_strlen = strlen('Orbit Radius (AU)')
    maxwidth = max_strlen * !d.x_ch_size + 6 ; ... + 6 for padding

    ; Create base widget, two tables (column- and row-major,
    ; respectively), and 'Quit' button.
    base = WIDGET_BASE(/COLUMN)
    table1 = WIDGET_TABLE(base, VALUE=data, /COLUMN_MAJOR, $
        ROW_LABELS=labels, COLUMN_LABELS='', $
        COLUMN_WIDTHS=maxwidth, /RESIZEABLE_COLUMNS)
    table2 = WIDGET_TABLE(base, VALUE=data, /ROW_MAJOR, $

```

```

        ROW_LABELS='', COLUMN_LABELS=labels, /RESIZEABLE_COLUMNS)
b_quit = WIDGET_BUTTON(base, VALUE='Quit', $
        EVENT_PRO='table_widget_example2_quit_event')

; Realize the widgets.
WIDGET_CONTROL, base, /REALIZE

; Retrieve the widths of the columns of the first table.
; Note that we must realize the widgets before retrieving
; this value.
col_widths = WIDGET_INFO(table1, /COLUMN_WIDTHS)

; We need the following trick to get the first column (which is
; a header column in our first table) to reset to the width of
; our data columns. The initial call to keyword COLUMN_WIDTHS
; above only set the data column widths.
WIDGET_CONTROL, table1, COLUMN_WIDTHS=col_widths[0], $
        USE_TABLE_SELECT=[-1,-1,3,3]
; This call gives table 2 the same cell dimensions as table 1
WIDGET_CONTROL, table2, COLUMN_WIDTHS=col_widths[0], $
        USE_TABLE_SELECT=[-1,-1,3,3]

; Call XMANAGER to manage the widgets.
XMANAGER, 'table_widget_example2', base

END

```

The following things about this example are worth noting:

- By default, column and row titles will contain the index of the column or row. To remove either column or row titles entirely, set the value of the `COLUMN_LABELS` or `ROW_LABELS` keyword to an empty string ('').
- Setting the width of the row-title column of the row-major table requires us to select column -1 using the `USE_TABLE_SELECT` keyword.



Chapter 8

Using Tab Widgets

The following topics are covered in this chapter:

Using Tab Widgets	178	Tab Sizing and Multiline Behavior	180
Example: A Simple Tab Widget	179	Example: Retrieving Values	182

Using Tab Widgets

Tab widgets create a “tabbed” interface that allows the user to select one of a list of rectangular display areas to be displayed in a single space (the *tab set*). The displayed interface elements are contained in base widgets — that is, selecting a tab displays the contents of a specified base widget within the tabbed interface. See “[WIDGET_TAB](#)” (*IDL Reference Guide*) for a complete description of the function used to create tab widgets.

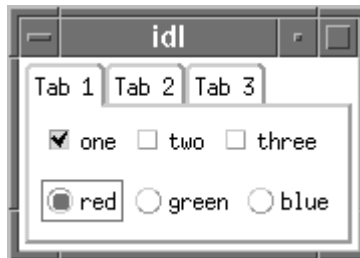


Figure 8-1: A tab widget displaying a tab set with three tabs.

Example: A Simple Tab Widget

The following procedures build a simple tabbed interface with three tabs containing a variety of other widgets.

Example Code

This example is included in the file `tab_widget_example1.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution. Run this example procedure by entering `table_widget_example1` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT table_widget_example1.pro`. See [“Running the Example Code”](#) on page 15 if IDL does not run the program as expected.

Calling `tab_widget_example1` with the `LOCATION` keyword set to an integer value between 0 and 4 displays the same interface with the tabs placed on different sides.

As with many of the examples in this chapter, this one is designed to merely exhibit the features of the tab widget. Most of the useful things you might do with a tab widget take place in the event handling routines for the individual widgets displayed on each tab; see [“Example: Retrieving Values”](#) on page 182 for a more complicated example that stores the values of the individual widgets for later use.

Tab Sizing and Multiline Behavior

The size of the rectangular area of the tab display (where individual widgets are placed) is determined by the size of the largest base widget included in the tab set. The size of the “tab” itself (the curved area that sticks out from the rectangular base and contains the tab’s title) is determined by a number of factors, including the size of other tabs, the presence of the `LOCATION` and `MULTILINE` keywords, and the platform on which the widget application is running.

IDL attempts to create a tab that is large enough to contain the tab’s title (which is set via the `TITLE` keyword to `WIDGET_BASE` for the base widget that has the tab widget as its parent). This, coupled with the fact that the value of the `MULTILINE` keyword has different meanings on different platforms (see “[WIDGET_TAB](#)” (*IDL Reference Guide*) for details), leads to the following behaviors:

Windows Behavior

Tabs are created to show the entire text of the `TITLE` keyword to `WIDGET_BASE`.

If `LOCATION = 0` or `1`

Setting the `LOCATION` keyword to `WIDGET_TAB` equal to zero places the tabs on the top of the tab set; setting `LOCATION` to one places the tabs on the bottom of the tab set. In either case, if the `MULTILINE` keyword is set equal to zero, and the width of the tabs exceeds the width of the largest child base widget, the tabs are shown with scroll buttons. This allows the user to scroll through the tabs while the base widget stays immobile.

If the `MULTILINE` keyword is set to a positive value, the tabs will be placed in as many rows as are necessary in order to display the entire text of each tab (limited by the width of the largest base, see note below).

If `LOCATION = 2` or `3`

Setting the `LOCATION` keyword to `WIDGET_TAB` equal to two places the tabs on the left edge of the tab set; setting `LOCATION` equal to three places the tabs on the right edge of the tab set. In either case, a multiline display is always used if the width of the tabs exceeds the height of the largest child base widget, even if the `MULTILINE` keyword is set equal to zero. Tabs are placed in as many rows as are necessary in order to display the entire text of each tab (limited by the height of the largest base, see note below).

Note

The width or height of the tab widget is based on the width or height of the largest base widget that is a child of the tab widget. If the width of the text of one tab exceeds the width or height of the tab widget, the text will be truncated even if the MULTILINE keyword is set.

Motif Behavior

Motif platforms interpret the value of the MULTILINE keyword to be the maximum number of tabs to display per row. If the keyword is not specified or is explicitly set equal to zero, all tabs are placed on the same row. Tabs are created to show the entire text of the TITLE keyword to WIDGET_BASE. The text of the tabs is not truncated in order to make the tabs fit the space available, unless the text of a single tab exceeds the width or height of the largest base widget that is a child of the tab widget. This means that if the MULTILINE keyword is set to any value other than one, some tabs may not be displayed.

Tips for Tab Layout

There is no good way to determine in advance the best setting for the MULTILINE keyword to ensure an appropriate tab display. In most cases, however, the following suggestions should enable you to create a tab display that is useful on both Windows and UNIX platforms.

- Keep tab titles short. If you need a long description of the contents of a tab, use a label widget in the tab's base widget rather than creating a long title.
- Set the MULTILINE keyword equal to a value greater than one. This allows you to tune the appearance of your tab set to the Motif platform without changing the appearance under Windows, since any value greater than zero will result in a multiline tab display under Windows.
- If practical, place the tabs along the longest dimension of the tab widget, as determined by the size of the largest base widget.

Example: Retrieving Values

The following example builds on “[Example: A Simple Tab Widget](#)” on page 179 by adding the following features:

- “Next” and “Previous” buttons that switch the tab display to the next (or previous) tab in the tab set.
- A mechanism for saving the values of the widgets in the tab interface. Implementing such a mechanism allows the user to view and change all of the settings accessible via the tab widget before committing any of them.
- A mechanism for canceling — exiting from the tabbed interface without committing any changes made via the tab interface.

Example Code

This example is included in the file `tab_widget_example2.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution. Run this example procedure by entering `table_widget_example2` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT table_widget_example2.pro`. See “[Running the Example Code](#)” on page 15 if IDL does not run the program as expected.

The following things about this example are worth noting:

- The `retStruct` structure is an example of the kind of information you might pass *out* of a tab widget, back to a larger widget application. Using an approach like the one here allows the user to set a group of values before sending any of them to the larger application. This may be more efficient than updating the larger application “on the fly” as the user makes changes to the widgets in the tab interface.
- Similarly, if the user’s changes are not sent to the larger application until he or she clicks the “Done” button, it is important to provide a way for the user to cancel the operation entirely, without sending any changes.
- In most cases, when we refer to a field in a structure, we refer to it by its *name*. The event function `TWE2_saveValue` refers to the fields of the `retStruct` structure by their *indices* instead. We do this because while it is not possible to pass the field name in a variable, it is possible to pass the integer index value. Passing the index value of the appropriate field in the `retStruct` structure as the user value of the widget whose value is being saved allows us to write a single `TWE2_saveValue` function, rather than one function for each field in the `retStruct` structure.

- The “Next” and “Previous” buttons in this example imply that there is an order to the actions performed using the tab set. While there is no order in this example, it is easy to imagine a situation in which values from one tab would influence actions taken on another. You could even *require* that some action be taken on a given tab before a later tab could be displayed.



Chapter 9

Using Tree Widgets

The following topics are covered in this chapter:

Using Tree Widgets	186	Replacing the Default Bitmaps	191
Types of Tree Widgets	187	Dragging and Dropping Tree Nodes	193
Example: A Simple Tree	188	Tree Widget Drag and Drop Examples ..	205
Setting the Tree Selection State	189	Positioning Tree Nodes	207
Making a Tree Entry Visible	190		

Using Tree Widgets

Tree widgets display information in a hierarchical structure or *tree*. Branches and sub-branches of the tree can be expanded and collapsed (either programmatically or by the user) to display or hide the information they contain. See “[WIDGET_TREE](#)” (*IDL Reference Guide*) for a complete description of the function used to create tree widgets.

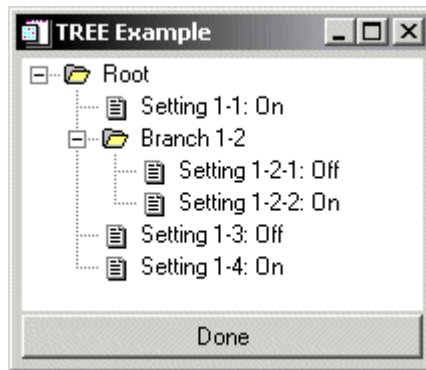


Figure 9-1: A tree widget.

This section discusses the following topics:

- “[Types of Tree Widgets](#)” on page 187
- “[Example: A Simple Tree](#)” on page 188
- “[Setting the Tree Selection State](#)” on page 189
- “[Making a Tree Entry Visible](#)” on page 190
- “[Replacing the Default Bitmaps](#)” on page 191
- “[Dragging and Dropping Tree Nodes](#)” on page 193
- “[Positioning Tree Nodes](#)” on page 207

Types of Tree Widgets

Tree widgets behave slightly differently depending on whether their parent widget is a base widget or another tree widget:

- If the tree widget's *Parent* is a base widget, the tree widget becomes the root of a new tree widget hierarchy. The tree widget itself is not displayed, but it becomes the parent widget for other tree widgets that *are* displayed. This type of tree widget is referred to as a *root node*. Note that a tree widget that is a root node cannot be the parent widget for any widget except another tree widget.
- If the tree widget's *Parent* is an existing tree widget, the new tree widget becomes a *branch node* or *leaf node* in the existing tree widget. In this case:
 - If the FOLDER keyword to WIDGET_TREE is present, the node becomes a *branch node*. Tree widgets that are branch nodes can become the parent widget of other tree widgets (but not of any other type of widget).
 - If the FOLDER keyword to WIDGET_TREE is *not* present, the node becomes a *leaf node*. Leaf nodes cannot serve as the parent widget for any other widget.

Example: A Simple Tree

The following example builds a simple tree widget (shown in [Figure 9-1](#)). Double-clicking on the leaf nodes toggles the value of the displayed text between the values “On” and “Off.”

Example Code

This example is included in the file `tree_widget_example.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution. Run this example procedure by entering `tree_widget_example` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT tree_widget_example.pro`. See [“Running the Example Code”](#) on page 15 if IDL does not run the program as expected.

As with many of the examples in this chapter, this one is designed to merely exhibit the features of the tree widget. Most of the useful things you might do with a tree widget take place in the event handling routines for the leaf nodes; whereas in this example clicking on a leaf simply changes the displayed text value, in a real application more complicated things might take place. Alternately, you might use a tree widget for display purposes only, in which case user interaction would be limited to expanding and collapsing the branches.

Setting the Tree Selection State

You can programmatically select or deselect nodes in a tree widget hierarchy using the `SET_TREE_SELECT` keyword to `WIDGET_CONTROL`. Selecting a node or nodes visually highlights the node on the tree display. In the above example, placing the following command just above the call to `XMANAGER`:

```
WIDGET_CONTROL, wtLeaf11, /SET_TREE_SELECT
```

would cause the first leaf node to be highlighted when the widget tree was first displayed.

Making a Tree Entry Visible

If your tree is large or has many branches, you may need to explicitly bring a given node to the user's attention. You can do this using the `SET_TREE_VISIBLE` keyword to `WIDGET_CONTROL`:

```
WIDGET_CONTROL, wTreeNode, /SET_TREE_VISIBLE
```

where *wTreeNode* is any node attached to a tree widget — that is, any tree widget that has another tree widget as its parent widget. Setting this keyword has two possible effects:

1. If the specified node is inside a collapsed folder, the folder and all folders above it are expanded to reveal the node.
2. If the specified node is in a portion of the tree that is not currently visible because the tree has scrolled within the parent base widget, the tree view scrolls so that the selected node is at the top of the base widget.

Use of this keyword does not affect the tree widget selection state.

Replacing the Default Bitmaps

By default, tree widgets use bitmap images of a folder and a single piece of paper as the icons representing branch and leaf nodes in a tree widget hierarchy. You can modify the look of the tree widget by supplying your own bitmap for a given node. Set the BITMAP keyword to WIDGET_TREE equal to a 16 x 16 x 3 array that contains a 24-bit color image.

For example, suppose you have a 16 x 16 pixel TrueColor icon stored in a TIFF file. The following commands make the image the icon used for the root node of a tree widget hierarchy:

```
myIcon = READ_TIFF('/path_to/myicon.tif', INTERLEAVE=2)
wtRoot = WIDGET_TREE(wTree, /FOLDER, BITMAP=myIcon)
```

Note the use of the INTERLEAVE keyword to ensure that the resulting image array has dimensions 16 x 16 x 3. Depending on your image file format, you may need to modify the image array in other ways.

Using Images from the IDL Distribution

The `/resources/bitmaps` subdirectory of the IDL distribution contains a selection of 16-color (4-bit), 16 x 16 pixel icon images. To use these images as bitmaps in a tree widget, you must convert them to 16 x 16 x 3 (24-bit color) arrays.

The following code snippet loads the camera icon stored in `IDLDIR/resources/bitmaps/camera.bmp` into a 16 x 16 x 3 array:

```
; Create a 24-bit image array.
imageRGB = BYTARR(16,16,3,/NOZERO)
; Read in the bitmap.
file=FILEPATH('camera.bmp', SUBDIR=['resource', 'bitmaps'])
image8 = READ_BMP(file, Red, Green, Blue)
; Pass the image through the color table
imageRGB[0,0,0] = Red[image8]
imageRGB[0,0,1] = Green[image8]
imageRGB[0,0,2] = Blue[image8]
```

To use the camera icon in a tree widget, you would specify the `imageRGB` variable as the value of the BITMAP keyword to WIDGET_TREE.

Adding Transparency to Icons

User supplied bitmaps can have transparent pixels, just as the default icons do. This ensures that the icon's background matches the background of your tree widget, which you can customize. It also enables one bitmap to suffice for all platforms, which often have different default tree widget background colors.

The MASK family of keywords is used to affect tree widget bitmaps. When a tree widget is created with a bitmap or given a new bitmap, the MASK keyword serves to set the masked sub-property of the bitmap. Those pixels in the bitmap that have the same color as the lower left pixel are made to be transparent. Internally, a mask is built and then used to draw only those non-transparent pixels.

The following code creates a tree widget node with an unmasked bitmap and then changes it to a different, masked bitmap.

```
node = WIDGET_TREE( parentNode, BITMAP = icon1 )
WIDGET_CONTROL( node, SET_TREE_BITMAP = icon2, /SET_MASK )
```

Note

The MASK keyword of WIDGET_CONTROL has no effect when not used with SET_TREE_BITMAP. The MASK keyword to WIDGET_INFO can be used to determine if a tree's bitmap is masked. For more information see [WIDGET_CONTROL](#) and [WIDGET_INFO](#) in the *IDL Reference Guide*.

Dragging and Dropping Tree Nodes

In IDL versions 6.3 and later, you can create applications that allow users to drag tree nodes within a single tree widget, between tree widgets, or from a tree widget to a draw widget. Depending on the circumstances, the dragged tree node is either copied to the new location (leaving the source node intact) or moved to the new location (removing the source node). IDL provides a variety of controls that allow you to define the exact behavior of the application when a user drags and drops tree nodes.

This section discusses the following topics:

- [“The Drag and Drop User Interface”](#) on page 194
- [“Implementing Drag and Drop Functionality”](#) on page 196
- [“Tree Widget Drag and Drop Examples”](#) on page 205

The Drag and Drop User Interface

To the user of an IDL program that supports drag and drop functionality, the activity of dragging and dropping conforms to platform guidelines. The user selects one or more nodes using the left mouse button and drags them while holding the mouse button down. When dragging a node, the cursor indicates where the drop is allowed (above, on, below) or not allowed. Optionally, the cursor can include a + symbol to indicate the different between copy and move operations.

Note

On Windows platforms, the cursor shows an opaque copy of the node under the mouse pointer, but does not show all selected nodes, even though they are selected and are dragged. On UNIX platforms, a cursor reflecting the active drag is all that is shown.

In addition to the default platform-specific drag and drop behavior, IDL tree widgets implement the following:

- If the tree widget includes a vertical scroll bar, dragging nodes into the region at the top or bottom of the widget will automatically scroll to bring new nodes into view.
- If the user has dragged one or more nodes to a new location, but presses the Escape key before releasing the mouse button, the drag operation is cancelled.

As the user drags the selected nodes, the drag and drop cursor changes to indicate whether a drop is allowed at the current position. The drag and drop cursor is displayed differently on different platforms:

Node Placement	Windows	UNIX
Above	Horizontal line above target node	Arrow bent and pointed up
On	Target node highlighted	Straight arrow
Below	Horizontal line below target node	Arrow bent and pointed down
Not Allowed	Circle with slash through	Circle with slash through

Table 9-1: Platform-Specific Appearance of the Drag and Drop Cursor

Figure 9-2 shows the appearance of the drag and drop cursor when inserting a node (here named *Leopard*) after a node named *Jaguar*, but within the node category *Spotted*.

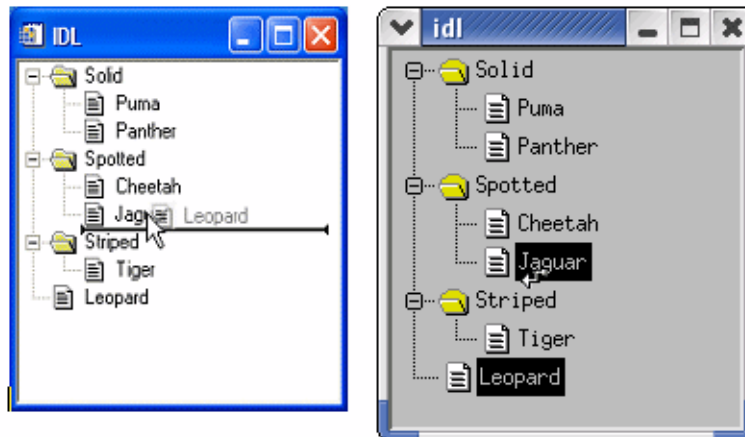


Figure 9-2: Inserting a Node After Another Node in Windows (left) and Unix (right)

Note

The IDL application ultimately controls which nodes to copy or move, where they are placed, and the destination tree's final state. For example, IDL may override the restoration of a previous selection and instead select newly copied nodes.

Implementing Drag and Drop Functionality

Drag and drop functionality is not enabled by default. When creating an IDL application that incorporates a tree widget, you can enable drag and drop behavior to copy, move, or otherwise rearrange tree widget nodes. This section discusses the steps necessary to implement drag and drop functionality in your application.

Implementing drag and drop functionality in your tree widget application entails three steps:

1. **Making Nodes Draggable.** You must explicitly specify that a node or group of nodes can be dragged.
2. **Responding to Drag Notifications (Callbacks).** When the user drags a node, IDL generates a *notification*, which is passed to a *callback function*. You can use the default callback function for simple situations, or create your own callback function to handle special or complex situations. Drag notifications allow you to control if and where drops are allowed.
3. **Responding to Drop Events.** When the user releases the mouse button to drop the selected nodes, IDL generates a *drop event*. You can use the information contained in the drop event structure to copy, move, or otherwise modify the tree widget.

Drag and Drop Properties are Inheritable

Drag and drop-related properties of a tree widget node (the values of the `DRAG_NOTIFY`, `DRAGGABLE`, and `DROP_EVENTS` keywords) are inheritable. This means that unless the value of one of these keywords is set specifically for a given tree node, that node will inherit the value of its parent. This means that if you set these values on the root node of a tree, but not on any child node, all nodes will have the values specified for the root node.

Inheritance is dynamic. This means that if the value of one of the inherited properties changes after the tree widget has been created (via a change of parent, due to a drag and drop operation, or via a call to `WIDGET_CONTROL`), the values for all of the inheriting nodes will change as well. One advantage of this type of inheritance is that nodes don't keep track of their own property settings as they are copied and moved. This allows you to create, for example, a folder that allows items to be dropped in, but not dragged out, simply by setting properties on the folder.

The drag and drop-related properties can all be queried using the `WIDGET_INFO` function.

Making Nodes Draggable

The value of the DRAGGABLE property of a tree widget node (as set via the DRAGGABLE keyword to `WIDGET_TREE` or the SET_DRAGGABLE keyword to `WIDGET_CONTROL`) determines whether or not it can be used to initiate drag and drop operations.

Note

The value of a tree node's druggability is independent of its dropability. Making a node draggable does make it droppable, but it is possible to have no allowable place to drop it. See “[Responding to Drag Notifications \(Callbacks\)](#)” on page 197 for information on allowing users to drop nodes.

If a tree widget allows multiple selection (if the MULTIPLE keyword was set on the root node of the tree), it is possible that a user could select a mixture of draggable and non-draggable nodes. If the user attempts to drag this mixed selection by moving a draggable node, your IDL application will have to determine whether to allow a drop. You have several possible options to respond to this situation:

- **Prevent the problem** — Prevent the user from creating a mixed selection by responding to selection events and then programmatically altering the selection to make it legal.
- **Deny all drops** — Use a drag notification callback to inspect the selection (the dragged items) and reject all drops if any of the selected items are non-draggable.
- **Allow the drag but only drop a subset of the nodes** — Create a routine that checks (and possibly modifies) the list of selected nodes before calling the `WIDGET_TREE_MOVE` routine. Alternately, create your own copy/move routine.

Responding to Drag Notifications (Callbacks)

When the user drags a group of selected nodes over another node, IDL automatically calls the routine defined as the *drag notification callback* for the node over which the selection is dragged. The purpose of the drag notification callback is to provide the widget system with information about where dragged nodes can be dropped, allowing it to change the cursor display to indicate to the user whether nodes can be dropped at the current position. You, as an IDL application programmer can choose to specify your own version of the callback function to override the default behavior. Drag notification callbacks are specified via the DRAG_NOTIFY keyword to `WIDGET_TREE`, or the SET_DRAG_NOTIFY keyword to `WIDGET_CONTROL`.

Drag notifications are also generated when the state of a drag modifier key changes (either up or down). If you override the default drag notification callback, you can use this information to update the drag cursor with a plus symbol (+).

You can specify a unique drag notification callback function for each node. If you choose not to specify a callback for a particular node, it will inherit the callback defined for its parent node. If no callback is defined for any of a particular node's ancestors, the default callback will be used.

Drag Notification Callback Return Values

The drag notification callback function returns an integer value calculated by ORing the following values together:

Value	Meaning
0	User cannot drop
1	User can drop above
2	User can drop onto
4	User can drop below
8	Show the plus indicator

Table 9-2: Drag Notification Callback Return Values

For example, if the drag notification callback returns 7, this means that dragged nodes can be dropped above, onto, or below the currently selected node. If the callback returns 10, the dragged nodes can be dropped onto (but not above or below) the current node, and the plus-sign indicator is included in the cursor.

The Default Drag Notification Callback

The default drag notification callback function is used if no function is specified for a given node or any of its ancestors. The return values for the default callback depend on the location of the node being targeted and (if it is a folder) whether it is expanded or not:

Tree Widget Node Type	Expanded	Return Value	Meaning
Root		2	Onto
Folder	No	7	Above, Onto, Below
	Yes	3	Above, Onto
Leaf		5	Above, Below

Table 9-3: Default Drag Notification Callback Return Values

The default callback also compares the dragged nodes with the destination. If the destination matches or is a descendant of any of the dragged nodes then the default callback returns 0. Finally, if the destination will not generate drop events (`DROP_EVENTS = 0`) then the default callback will return 0.

Writing Custom Drag Notification Callbacks

The signature of a drag notification callback function is:

```
FUNCTION Callback_Function_Name, Destination, Source, $
    Modifiers, Default
```

where:

- *Callback_Function_Name* is the name of the callback function
- *Destination* is the widget ID of the node over which the drag cursor is currently positioned
- *Source* is the widget ID of the source tree, from which a list of widget IDs representing the list of selected nodes can be retrieved using the `TREE_SELECT` or `TREE_DRAG_SELECT` keywords to `WIDGET_INFO`.
- *Modifiers* is an integer value calculated by ORing the following values together, depending on which modified keys are currently depressed:

Value	Modifier Key
1	Shift
2	Control
4	Caps Lock
8	Alt

Table 9-4: Modifier keys

- *Default* is the value that would be returned by the default drag notification callback. If your criteria are similar to the default criteria for where nodes can be dropped, using this value can greatly simplify your callback code.

The return value should indicate where a drop is allowed to take place relative to the destination widget and whether the “+” symbol should appear with the drag cursor, as described in [Table 9-2](#).

When you write drag notify callbacks, remember that “above” one node may be “below” another node. Also, the concepts of “above,” “on,” and “below” are relative to the level of the destination node. For example, if a node is the final (bottom) node, then its defined “below” is a different position than it would be for its parent. The default callback takes these differences into account and does not allow you to drop below an open folder, preventing confusion over whether the dropped nodes will get into or below the folder.

When writing drag notification callbacks, keep the following in mind:

1. Drag callbacks should execute quickly. If a callback takes too long to drag, events may be skipped. Remember that the drag callback is invoked after every change in position of the cursor.
2. The source and destination trees should not be modified during the drag. Callbacks should not select, unselect, create, move, or delete nodes of the source or destination trees. Additionally, layout changes affecting the trees are also strongly discouraged.
3. The drag callback should be tested thoroughly using a CATCH statement. Although the widget system will do its best to recover from errors that occur in a drag callback function, errors inside the callback function could lead to loss of keyboard and mouse input.

Note

In Windows recovery an error in the callback function is simple: click away from IDL and then back on IDL. Recovery on UNIX systems may require that the IDL session be killed from another terminal session.

The following code shows a callback function that intentionally generates an error, along with a CATCH statement that can be used to prevent the error from freezing IDL:

```
FUNCTION bad_callback, dest, source, modifiers, default

; The following CATCH statement protects against UI freezes.
CATCH, Error_status
IF Error_status NE 0 THEN BEGIN
    CATCH, /CANCEL
    PRINT, 'Error index: ', Error_status
    PRINT, 'Error message: ', !ERROR_STATE.MSG
    RETURN, 0
ENDIF

; The undefined variable caused an IDL interpreter error.
IF (undefined EQ 0) THEN RETURN, 7 $
ELSE RETURN, 0

END
```

In this example, an error occurs because the variable `undefined` is undefined. The catch block handles this error and prevents loss of keyboard and mouse control.

You can also test your callback functions by explicitly calling them before the widget system does. This would test the callbacks from a safe state where the implications of errors are minor.

“[Tree Widget Drag and Drop Examples](#)” on page 205 shows several uses of the default and custom callbacks. All of these examples have reliable static callbacks, allowing for safe removal of CATCH statements.

Responding to Drop Events

When the user releases the mouse button over a valid drop target, a `WIDGET_DROP` event is generated. Your application’s event handler should recognize this drop event and perform some action. In most cases, the event handler will call code to move or copy the selected nodes (see the [WIDGET_TREE_MOVE](#) routine for a ready-made move/copy routine), but you can execute any action you wish when the drop event is generated.

The drop event's information is contained in a `WIDGET_DROP` structure. (See [Drop Events](#) in the reference section for `WIDGET_TREE` for a full definition of the `WIDGET_DROP` structure.) The important components of the structure when responding to drop events are:

- **ID** — The widget ID of the destination node. You can use the `INDEX` keyword to `WIDGET_INFO` along with this widget ID to determine the index of the destination node within the tree widget.
- **DRAG_ID** — The widget ID of the source tree widget. The selected nodes of this tree are the nodes that are being dragged. You can use the `TREE_DRAG_SELECT` and `TREE_SELECT` keywords to `WIDGET_INFO` along with this widget ID to retrieve the list of selected nodes or `TREE_DRAG_SELECT`.
- **POSITION** — The drop position (above, on, or below) relative to the drop target (returned in the `ID` field). Use this value, along with index of the destination node, to determine the index of the location where the dropped nodes should be inserted.
- **MODIFIERS** — An integer representing the state of the modifier keys, calculated by ORing together the values shown in [Table 9-4](#). On some platforms it is common for the **Ctrl** key to be used as the copy key, with simple move operations being performed when **Ctrl** is not pressed.

Issues Related to Dropping Nodes

IDL's drag and drop functionality is quite general, because applications can have diverse requirements. Trees might allow only a single node to be selected, or may allow multiple selection. The application might use the **Ctrl** key to distinguish between copy and move operations. Other drag and drop issues that need to be solved by your specific application include:

- **Copying nodes that are not marked as DRAGGABLE** — IDL's widget system does not mandate what can or will be copied. The `DRAGGABLE` keyword controls only the initiation of dragging. Applications can choose not to copy any node that is not `DRAGGABLE`.
- **Dragging a node to one of its descendants** — The default drag notification callback invalidates all drops that occur on a drag source or any of the drag source's descendants. If you write your own drag notification callback, be sure to reject drops onto a source node (or any of its descendants) to avoid infinite recursion.
- **Copying unselected children of selected parents** — This is shown in the following figure.

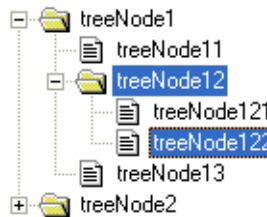


Figure 9-3: Copying Unselected Children of Selected Parents

If **treeNode12** is dragged and dropped, should **treeNode121** also be copied, if we know that **treeNode122** has been specifically selected and **treeNode121** has not? One solution to this could be an individual copy of only the selected nodes. Another solution could be to attempt to preserve the hierarchy. Also, an application could choose to use the drag callback to reject the selection as unsuitable for dropping anywhere. The examples used here assume that a folder is dragged and dropped because all descendents are wished to be copied along with that folder, regardless of the selection state.

The following code illustrates one way to handle drop events:

```

PRO handle_drop_event, event

; figure out the new node's parent and the index
;
; The key to this is to know whether or not the drop took
; place directly on a folder. If it was then the new node
; will be created within the folder as the last child.
; Otherwise the new node will be created as a sibling of
; the drop target and the index must be computed based on
; the index of the destination widget and the position
; information (below or above/on).

IF (( event.position EQ 2 && $
      WIDGET_INFO( event.id, /TREE_FOLDER )) THEN BEGIN

    wParent = event.id
    index = -1

ENDIF ELSE BEGIN

    wParent = WIDGET_INFO( event.id, /PARENT )
  
```

```

        index = WIDGET_INFO( event.id, /TREE_INDEX )
        IF ( event.position EQ 4 ) THEN index++

    ENDELSE

; move the dragged node (single selection tree)

wDraggedNode = WIDGET_INFO( event.drag_id, /TREE_SELECT )

WIDGET_TREE_MOVE, wDraggedNode, wParent, INDEX = index

END

```

This code does the following things:

- **Determines the parent and insertion position (index) for the new node** — Drops can be above, on, or below, and the destination node can be a folder or a leaf. This example determines where to place the new node. Dropping onto a folder is the simplest option, but other situations require a knowledge of where the destination sits relative to its siblings. The INDEX group of tree widget keywords allows us to query and set the position of tree widget nodes. For more information on these keywords, see “[WIDGET_TREE](#)” (*IDL Reference Guide*).
- **Moves the selected node to the new position** — First, we determine the widget ID of the dragged node, then use the [WIDGET_TREE_MOVE](#) procedure to move it to the new location.

The above example works well for single selection trees that use the default drag notification callback. Situations involving multiple selection should use the TREE_DRAG_SELECT keyword to [WIDGET_INFO](#) rather than TREE_SELECT.

A more complete version of the previous example and more complex examples involving multiple selection and custom callbacks can be found in the next section.

Tree Widget Drag and Drop Examples

Tree widget drag and drop scenarios can range from relatively straight forward to quite complex. The degree of complexity usually increases with the inclusion of multiple selection and custom drag callbacks. Two example applications are included in the IDL distribution, illustrating simple and relatively complex drag and drop applications.

Simple Drag and Drop Example

The IDL distribution contains an example that creates a single selection tree that allows you to rearrange the folder and leaf nodes. The example demonstrates how to:

- Enable dragging and drop events
- Handle drop events
- Move a node in response to a drop event
Including:
 - Determining the insertion parent
 - Determining the insertion index

Example Code

The simple drag and drop example is included in the file `drag_and_drop_simple.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution. Run this example procedure by entering `drag_and_drop_simple` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT drag_and_drop_simple.pro`.

Complex Drag and Drop Example

The IDL distribution contains an example that creates three multiple selection trees and demonstrates how to:

- Handle multiple selection drags
- Create custom drag notification callbacks
- Implement control-key copying (versus simple moving)
- Implement “trigger-loaded” folders
- Enable dragging
- Enable drop events

- Copy one or more nodes in response to a drop event
Including:
 - Determining the insertion parent.
 - Determining the insertion index

This example also demonstrates many of the tree widget manipulation capabilities, such as those involving node indexes and masked bitmaps,

Example Code

The complex drag and drop example is included in the file `drag_and_drop_complex.pro` in the `examples/doc/widgets` subdirectory of the IDL distribution. Run this example procedure by entering `drag_and_drop_complex` at the IDL command prompt or view the file in an IDL Editor window by entering `.EDIT drag_and_drop_complex.pro`.

Positioning Tree Nodes

The position of tree widget nodes within a tree can be controlled in several ways. When a tree node is created without the `TOP` or `INDEX` keywords, the default behavior is to position the new node last among its siblings. If the `TOP` keyword is specified, the new node is created as the first child. If the `INDEX` keyword is specified, the new node is created at the position specified.

Existing tree widget nodes can be repositioned using the `SET_TREE_INDEX` keyword to `WIDGET_CONTROL`. Use this keyword for reordering nodes within an existing tree.

Note

The root node cannot be reordered relative to its siblings.

When positioning tree nodes using the `INDEX` or `SET_TREE_INDEX` keywords, the value is the desired zero-based index of the node. Values that are less than zero, or greater than or equal to the number of children will position the node as the last child.

You can use the `TREE_INDEX` keyword to `WIDGET_INFO` to discover the current position of a node. To get the node to a particular position, there are two additional useful keywords to `WIDGET_INFO`: the `N_CHILDREN` and `ALL_CHILDREN` keywords return the number and identifiers of a parent's children. For example, a tree widget can be “walked” as follows:

```
children=WIDGET_INFO( node, /ALL_CHILDREN )
FOR i=0, WIDGET_INFO( node, /N_CHILDREN ) - 1 DO BEGIN
; do something important here with children[i]
ENDFOR
```




Index

A

accelerators

Alt key on Mac, [93](#)

assigning, [92](#)

ignoring, [97](#)

application state

preserving, [42](#)

widgets, [42](#)

B

base widgets

bulletin board bases, [77](#)

setting size/location, [77](#)

bitmaps

transparent

button widgets, [103](#)

blocking, widgets, [35](#)

button widgets

about, [102](#)

accelerators, [92](#)

accelerators on Mac, [93](#)

checkbox, [107](#)

exclusive (radio), [107](#)

labels, [103](#)

nonexclusive (checkbox), [107](#)

push button, [107](#)

radio button, [107](#)

tabbing, [85](#)

toggle, [107](#)

tooltips, [106](#)

buttons

checkbox, [107](#)

radio, [107](#)

C

- callback routines
 - widget, 38
- callbacks
 - event processing, 38
- changing
 - widget values, 27
- checkbox widgets
 - See also* button widgets
- common blocks
 - widgets, 42
- compound widgets
 - creating, 46
- context_draw_example.pro, 75
- context_list_example.pro, 75
- context_menu_example.pro, 75
- context_text_example.pro, 75
- context_tlbase_example.pro, 75
- context-sensitive menu
 - about, 69
- controls *see* widgets
- copyrights, 2
- CW_DICE function, 49
- cw_dice.pro, 49
- CW_PDMENU function
 - creating menus, 67

D

- debugging
 - widget applications, 53
- destroying
 - widgets, 27
- displaying
 - tree widget section, 190
 - widgets, 19
- doc_widget1.pro, 21
- doc_widget2.pro, 40
- drag and drop
 - in tree widgets, 121, 193

- drag notifications
 - responding to, 121, 197, 201
- drag_and_drop_complex.pro, 206
- drag_and_drop_draw.pro, 124
- drag_and_drop_simple.pro, 205
- draw widgets
 - button events, 119
 - context events, 117
 - direct graphics, 111
 - keyboard events, 119
 - motion events, 119
 - object graphics, 112
 - scrolling, 113
 - using, 110
 - wheel events, 119
- draw_app_scroll.pro, 116
- draw_widget_data.pro, 119
- draw_widget_example.pro, 118
- drop events
 - handling, 123, 201
- droplist widgets
 - tabbing, 85

E

- event driven programming, 15
- event processing (widget applications), 34
- event processing *see* widgets, event processing
- events
 - interrupting the event loop, 56
 - See also* widget events
- examples
 - widgets
 - context_draw_example.pro, 75
 - context_list_example.pro, 75
 - context_menu_example.pro, 75
 - context_text_example.pro, 75
 - context_tlbase_example.pro, 75
 - cw_dice.pro, 49
 - doc_widget1.pro, 21
 - doc_widget2.pro, 40

drag_and_drop_complex.pro, 206
 drag_and_drop_draw.pro, 124
 drag_and_drop_simple.pro, 205
 draw_app_scroll.pro, 116
 draw_widget_data.pro, 119
 draw_widget_example.pro, 118
 tab_widget_example1.pro, 179
 tab_widget_example2.pro, 182
 table_widget_example1.pro, 170
 table_widget_example2.pro, 174
 tree_widget_example1.pro, 188
 xdice.pro, 52
 exclusive buttons *see* widgets, button

G

geometry
 widgets, 76

I

instantiating widgets, 19
 interrupt
 widget event loop, 56

K

keyboard
 accelerators, 92
 killing widgets, 27

L

legalities, 2
 location
 widgets, 77

M

Macintosh

 configuring accelerators, 93

mapping
 widgets, 27

menus

 context-sensitive, 69

 creating, 64

 creating pulldown, 66

message URL javascript

 doIDL(".edit context_menu_example.pro"),
 75

 doIDL(".edit cw_dice.pro"), 49, 50

 doIDL(".edit drag_and_drop_complex.pro"),
 206

 doIDL(".edit drag_and_drop_draw.pro"),
 124

 doIDL(".edit drag_and_drop_simple.pro"),
 205

 doIDL(".edit draw_app_scroll.pro"), 116

 doIDL(".edit draw_widget_data.pro"), 119

 doIDL(".edit draw_widget_example.pro"),
 118

 doIDL(".edit table_widget_example1.pro"),
 170, 179

 doIDL(".edit table_widget_example2.pro"),
 174, 182

 doIDL(".edit tree_widget_example.pro"),
 188

 doIDL(".edit xdice.pro"), 52

 doIDL("context_menu_example"), 75

 doIDL("cw_dice"), 49, 50

 doIDL("drag_and_drop_complex"), 206

 doIDL("drag_and_drop_draw"), 124

 doIDL("drag_and_drop_simple"), 205

 doIDL("draw_app_scroll"), 116

 doIDL("draw_widget_data"), 119

 doIDL("draw_widget_example"), 118

 doIDL("table_widget_example1"), 170, 179

 doIDL("table_widget_example2"), 174, 182

 doIDL("tree_widget_example"), 188

doIDL("xdice"), [52](#)

N

nonexclusive buttons *see* widgets, buttons

P

parent widget

about, [19](#)

pop-up menus *see* context-sensitive menus

properties

registering, [127](#)

property sheet widgets

changing properties, [131](#)

selecting properties, [128](#)

sizing, [134](#)

user-defined properties, [133](#)

using, [126](#)

R

radio buttons

See also button widgets, radio buttons

realizing widgets, [26](#)

registering

properties, [127](#)

retrieving

widget values, [27](#)

S

screen size

finding, [80](#)

selection

tree widgets, [189](#)

selection modes (table widget), [156](#)

sensitizing widgets

about controlling, [28](#)

shortcut menus *see* context-sensitive menus

size

of widgets, [77](#)

sizing

property sheets, [134](#)

T

tab widgets

sizing, [180](#)

using, [178](#)

tab_widget_example1.pro, [179](#)

tab_widget_example2.pro, [182](#)

table widgets

default size, [155](#)

edit mode, [162](#)

retrieving data, [159](#)

selection modes, [156](#)

tabbing, [85](#)

using, [154](#)

table_widget_example1.pro, [170](#)

table_widget_example2.pro, [174](#)

tooltips, [106](#)

trademarks, [2](#)

transparent bitmaps

button widgets, [103](#)

tree nodes

making draggable, [197](#)

positioning, [207](#)

tree widgets

drag and drop

about, [193](#)

drag notifications, [197](#)

drop events, [201](#)

enabling, [196](#)

interface, [194](#)

examples

drag and drop, [205](#)

simple, [188](#)

positioning nodes, [207](#)

replacing default bitmaps, [191](#)

- selection state, 189
- tabbing, 86
- types, 187
- using, 186
- visibility, 190

tree_widget_example1.pro, 188

U

- user interface
 - application options, 8
- user values (widgets), 33

W

- widget
 - visibility, 27
- widget events
 - about, 34
- widget values, 18
- WIDGET_CONTROL procedure
 - in widget applications, 26
 - manage widget manipulation, 28
- WIDGET_EVENT function
 - description, 29
 - when to use, 37
- WIDGET_INFO function
 - in widget manipulation, 29
- WIDGET_PROPERTY SHEET function
 - using, 126
- widgets
 - about, 14
 - aligning, 77
 - application state, 42
 - applications
 - defined, 15
 - errors, 53
 - lifecycle, 23
 - changing values, 27
 - common blocks, 42

- controlling visibility
 - overview, 27
- creating, 19
- destroying, 27
- displaying
 - in applications, 19
- draw
 - See* draw widgets.
- dynamic resizing, 77
- event processing, 38
 - concepts, 34
 - context events, 60
 - identifying widget types, 57
 - interrupting event loop, 56
 - keyboard focus, 57
 - techniques, 56
 - timer events, 58
 - tracking events, 59
- event structure, 31
- events
 - callback routines, 38
 - structure of, 34
- example code, 15
- explicit size, 76
- finding screen size, 80
- geometry, 76
- hierarchies, 26
- hierarchies, multiple, 61
- hourglass cursor, 28
- IDs
 - concept, 18
- instantiating, 19
- interrupting the event loop, 56
- killing, 27
- location, 77
- managing the state of applications, 42
- manipulating, 26
- mapping, 27
- menus
 - context-sensitive, 69
 - creating, 64

- pulldown, [66](#)
- natural size, [76](#)
- overview, [10](#), [14](#)
- parent, [19](#)
- portability, [83](#)
- positioning, [77](#)
- preventing layout flicker, [80](#)
- realizing
 - hierarchies, [26](#)
- restarting after an error, [53](#)
- retrieving values, [27](#)
- sensitivity, [28](#)
- sensitizing, [28](#)
- size
 - concepts, [76](#)
 - defining, [77](#)
 - dynamic resizing, [77](#)
 - natural, [76](#)
- tabbing, [84](#)
- types, [16](#)
- user values, [33](#)
- values, [18](#)
- widget IDs
 - working with, [31](#)
- WIDGET_CONTROL procedure, [26](#)
- WIDGET_EVENT function
 - in widget manipulation, [29](#)
 - when to use, [37](#)

- WIDGET_INFO function, [29](#)
- writing applications, [15](#)
- XMANAGER procedure
 - managing widget events, [29](#)
 - using, [35](#)
- XREGISTERED function
 - checking widget registration, [30](#)
 - using, [37](#)
- windows
 - finding screen size, [80](#)
- writing
 - a compound widget, [49](#)

X

- XBM_EDIT procedure
 - use of, [104](#)
- XDICE procedure, [51](#)
- xdice.pro, [52](#)
- XMANAGER procedure
 - managing widget events, [35](#)
 - overview, [29](#)
 - when to use XREGISTERED, [37](#)
- XREGISTERED function
 - using, [37](#)
 - widget registration, [30](#)