

Controls and Control Panels

- Overview 359
- Modes of Operation 359
- Using Controls 360
 - Buttons 360
 - Charts 360
 - Checkboxes 361
 - CustomControl 361
 - GroupBox 361
 - ListBox 361
 - Pop-Up Menus 361
 - Set Variable 362
 - Set Variable Controls and Data Folders 362
 - Sliders 362
 - TabControl 362
 - TitleBox 363
 - Value Displays 363
- Creating Controls 363
 - General Command Syntax 365
 - Button 365
 - Button Example 367
 - Custom Button Example 368
 - Charts and FIFOs 369
 - CheckBox 370
 - CustomControl 371
 - GroupBox 374
 - ListBox 374
 - PopupMenu 375
 - SetVariable 376
 - Slider 377
 - TabControl 378
 - TitleBox 380
 - ValDisplay 380
 - Numeric Readout Only 381
 - LED Display 381
 - Bar Only 381
 - Numeric Readout and Bar 381
 - Optional Limits 381
 - Optional Title 382
- Killing a Control 382
- Getting Information About a Control 382
- Updating a Control 382
- Help Text for User-Defined Controls 382
- Modifying a Control 383
- Disabling and Hiding a Control 383

Chapter III-14 — Controls and Control Panels

Background Color	383
Control Structures	384
Control Structure Example	385
Control Structure eventMod Field	385
Control Structure blockReentry Field	386
Control Structure blockReentry Advanced Example	386
User Data for Controls	387
Control User Data Examples	387
Action Procedures for Multiple Controls	388
Controls in Graphs	388
Drawing Limitations	389
Updating Problems	389
Control Panels	389
Embedding into Control Panels	390
Exterior Subwindows	390
Floating Panels	390
Control Panel Preferences	390
Controls Shortcuts	392

Overview

We use the term *controls* for a number of user-programmable objects that can be employed by Igor programmers to create a graphical user interface for Igor users. We call them *controls* even though some of the objects only display values. The term *widgets* is sometimes used by other application programs, especially on non-Macintosh systems.

Here is a summary of the types of controls available.

Control Type	Control Description
Button	Calls a procedure that the programmer has written.
Chart	Emulates a mechanical chart recorder. Charts can be used to monitor data acquisition processes or to examine a long data record. Programming a chart is quite involved.
CheckBox	Sets an off/on value for use by the programmer's procedures.
CustomControl	Custom control type. Completely specified and modified by the programmer.
GroupBox	An organizational element. Groups controls with a box or line.
ListBox	Lists items for viewing or selecting.
PopupMenu	Used by the user to choose a value for use by the programmer's procedures.
SetVariable	Sets and displays a numeric or string global variable. The user can set the variable by clicking or typing. For numeric variables, the control can include up/down buttons for incrementing/decrementing the value stored in the variable.
Slider	Duplicates the behavior of a mechanical slider. Selects either discrete or continuous values.
TabControl	Selects between groups of controls in complex panels.
TitleBox	An organizational element. Provides explanatory text or message.
ValDisplay	Presents a readout of a numeric expression which usually references a global variable. The readout can be in the form of numeric text or a thermometer bar or both.

The programmer can specify a procedure to be called when the user clicks on or types into a control. This is called the control's *action procedure*. For example, the action procedure for a button may interrogate values in pop-up menu, checkbox, and SetVariable controls and then perform some action.

Control panels are simple windows that contain these controls. These windows have no other purpose. You can also place controls in graph windows and in panel panes embedded into graphs. Controls are not available in any other window type such as tables, notebooks, or layouts. When used in graphs, controls are not considered part of the *presentation* and thus are **not** included when a graph is printed or exported.

Nonprogrammers will want to skim only the Modes of Operation and Using Controls sections, and skip the remainder of the chapter. Igor programmers should study the entire chapter.

Modes of Operation

With respect to controls, there are two modes of operation: one mode to use the control and another to modify it. To see this, choose Show Tools from the Graph or Panel menu. Two icons will appear in the top-left corner window. When the top icon is selected, you are able to use the controls. When the next icon is selected, the draw tool palette appears below the second icon. To modify the control, select the arrow tool from the draw tool palette.

When the top icon is selected or when the icons are hidden, you are in the *use* or *operate* mode. You can momentarily switch to the *modify* or *draw* mode by pressing Command-Option (Macintosh) or Ctrl+Alt (Windows). Use this to drag or resize a control as well as to double-click it. Double-clicking



Chapter III-14 — Controls and Control Panels

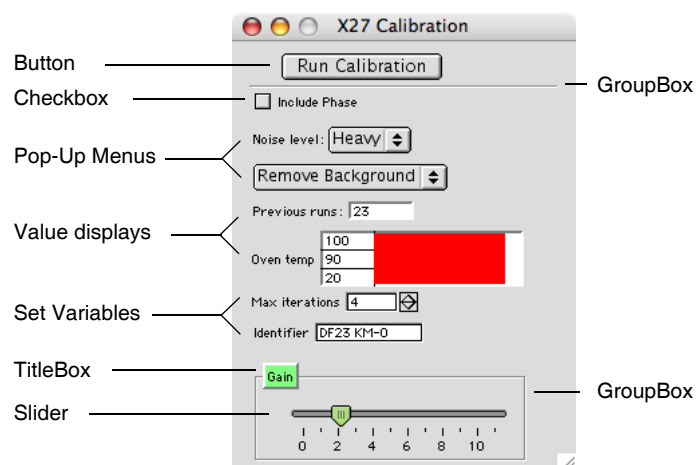
with the Command-Option (*Macintosh*) or Ctrl+Alt (*Windows*) pressed brings up a dialog that you use to modify the control.

You can also switch to modify mode by choosing an item from the Select Control submenu of the Graph or Panel menu.

Important: To enable the Add Controls submenu in the Graph and Panel menus, you must be in modify mode; either by clicking the second icon or by pressing Command-Option (*Macintosh*) or the Ctrl+Alt (*Windows*) while choosing the Add Controls submenu.

Using Controls

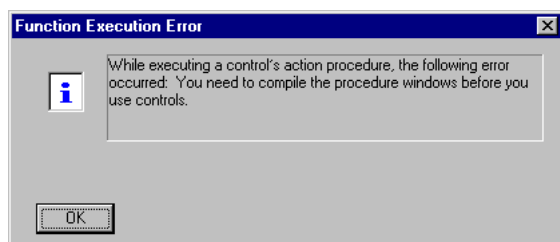
The following panel window illustrates most of the control types.



Buttons

When you click a button, it runs whatever procedure the programmer may have specified.

If nothing happens when you click a button, then there is no procedure assigned to the button. If the procedure window(s) haven't been compiled, clicking a button that has an assigned procedure will produce this dialog:



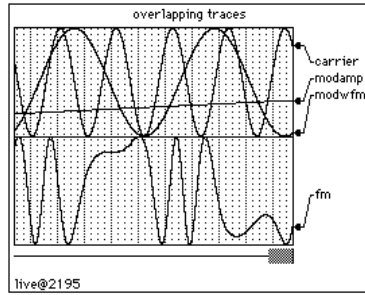
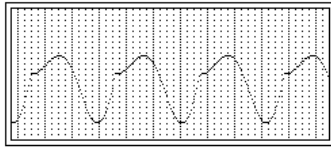
You should choose Compile from the Macros menu to correct this situation. If no error occurs then the button will now be functional.

Buttons usually have a rounded appearance, but a programmer can assign a custom picture so that the button can have nearly any appearance.



Charts

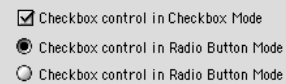
Chart controls can be used to emulate a mechanical chart recorder that writes on paper with moving pens as the paper scrolls by under the pens. Charts can be used to monitor data acquisition processes or to examine a long data record.



Note: Although programming a chart is quite involved, using a chart is actually very easy. However, since most users will never use a chart control, their use is described in **Charts and FIFOs** on page III-369.

Checkboxes

Clicking a checkbox changes its selected state and may run a procedure if the programmer specified one. A checkbox may be connected to a global variable. Checkboxes can be configured to look and behave like radio buttons.

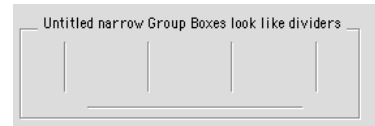


CustomControl

CustomControls are used to create completely new types of controls that are custom-made by the programmer. You can define and control the appearance and all aspects of a custom control's behavior. See **Custom-Control** on page III-371 for examples.

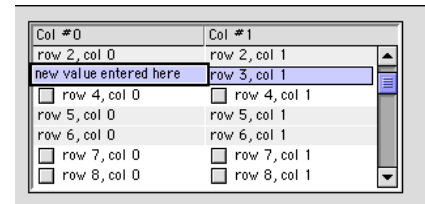
GroupBox

GroupBox controls are organizational or decorative elements. They are used to graphically group sets of controls. They may either draw a box or a separator line and can have optional titles.



ListBox

ListBox controls can present a single or multiple column list of items for viewing or selection. ListBoxes can be configured for a variety of selection modes. Items in the list can be made editable and can be configured as checkboxes.



Pop-Up Menus

These controls come in two forms: one where the current item is shown in the pop-up menu box



and another where there is no current item and a title is shown in the box.



The first form is usually used to choose one of many items while the second is used to run one of many commands.

Pop-up menus can also be configured to act like Igor's color, line style, pattern, or marker pop-up menus; these always show the current item.



Set Variable

Set Variable controls also can take on a number of forms and can display numeric values. Unlike Value Display controls that display the value of an expression, Set Variable controls are connected to individual global variables and can be used to set or change those variables in addition to reading out their current value. Set Variable controls can also be used with global string variables to display or set short one line strings. Set Variable controls are automatically updated whenever their associated variables are changed.

When connected to a numeric variable, these controls can optionally have up or down arrows that increment or decrement the current value of the variable by an amount specified by the programmer. Also, the programmer can set upper and lower limits for the numeric readouts.

New values for both numeric and string variables can be entered by directly typing into the control. If you click the control once you will see a thick border form around the current value.



You can then edit the readout text using the standard techniques including Cut, Copy, and Paste. If you want to discard changes you have made, press Escape. To accept changes, press Return, Enter, or Tab or click anywhere outside of the control. Tab enters the current value and also takes you to the next control if any. Shift-Tab is similar but takes you to the previous control if any.

If the control is connected to a numeric variable and the text you have entered can not be converted to a number then a beep will be emitted when you try to enter the value and no change will be made to the value of the variable. If the value you are trying to enter exceeds the limits set by the programmer then your value will be replaced by the nearest limit.

When a numeric control is selected for editing, the Up and Down Arrow keys on the keyboard act like the up and down buttons on the control.

Changing a value in a Set Variable control may run a procedure if the programmer has specified one.

Set Variable Controls and Data Folders

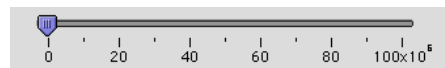
Note: If you are not using Data Folders (described in Chapter II-8, **Data Folders**), you can skip this section.

Set Variable controls remember the data folder in which the variable exists, and continue to function properly when the current data folder is different than the controlled variable. See **Set Variable** on page III-362.

The system variables (K0 through K19) belong to no particular data folder (they are available from any data folder), and there is only *one* copy of these variables. If you create a SetVariable controlling K0 while the current data folder is "aFolder", and another SetVariable controlling K0 while the current data folder is "bFolder", *they are actually controlling the same K0*.

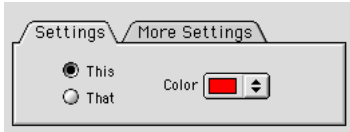
Sliders

Slider controls can be used to graphically select either discrete or continuous values. When used to select discrete values, a slider is similar to a pop-up menu or a set of radio buttons. Sliders can be live, updating a variable or running a procedure as the user drags the slider, or they can be configured to wait until the user finishes before performing any action.



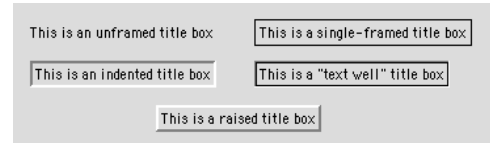
TabControl

TabControl are used to create complex panels containing many more controls than would otherwise fit. When the user clicks on a tab, the programmers procedure runs and hides the previous set of controls while showing the new set.



TitleBox

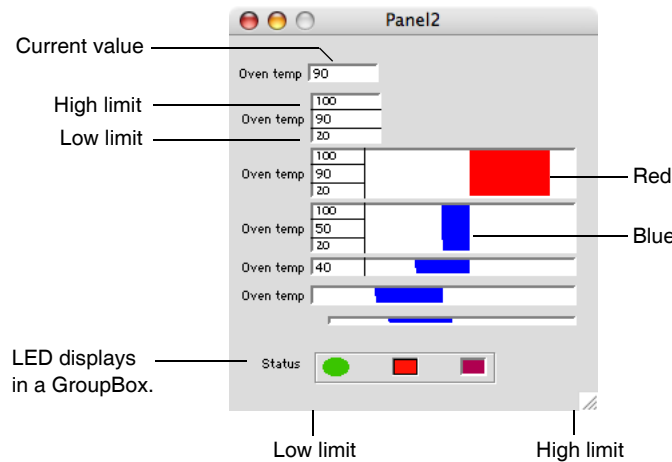
TitleBox controls are mainly decorative elements. They are used to provide explanatory text in a control panel. They may also be used to display textual results. The text can be unchanging, or can be the contents of a global string variable. In either case, the user can't inadvertently change the text.



Value Displays

These can take on a number of forms ranging from a simple numeric readout to a thermometer bar. Regardless of the form, value displays are just readouts. There is no interaction with the user. They display the current value of whatever expression the programmer specified. Often this will be just the value of a numeric variable, but it can be any numeric expression including calls to user-defined functions and external functions.

Here is a sampling of the forms that Value Display controls can assume.



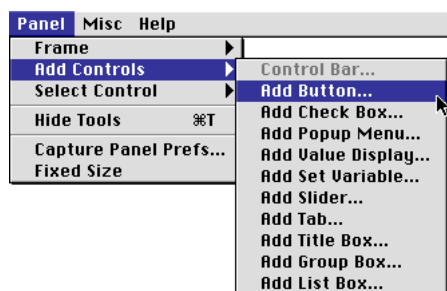
When a thermometer bar is shown, the left edge of the thermometer region represents a low limit set by the programmer while the right edge represents a high limit. The low and high limits appear in some of the above examples. The bar is drawn from a nominal value set by the programmer and will be red if the current value exceeds the nominal value and will be blue if it is less than the nominal value. In the above examples the nominal value is 60. There is no numeric indication of the nominal value. If the nominal value is less than the low limit then the bar will grow from the left to the right. If the nominal value is greater than the high limit then the bar will grow from the right to the left.

If you carefully observe a thermometer bar that is connected to an expression whose value is slowly changing with time you will see that the bar is drawn in a zig-zag fashion. This provides a much finer resolution than if the bar were to be extended or contracted by an entire column of screen pixels at once.

Creating Controls

The ease of creating the various controls varies widely. Anyone capable of writing a simple macro can create Buttons and Checkboxes, but creating Charts and CustomControls requires more expertise. Most controls can be created and modified using dialogs. You will find these dialogs under the Add Controls submenu in the Graph or Panel main menu.

Chapter III-14 — Controls and Control Panels



The Add Controls and Select Control menus are disabled until the arrow tool in the toolbar is enabled. (You use the toolbar's arrow tool to position and resize controls.) To do this, choose Show Tools from the Graph or Panel main menu and then click the second icon from the top (in the graph or panel's tool bar).

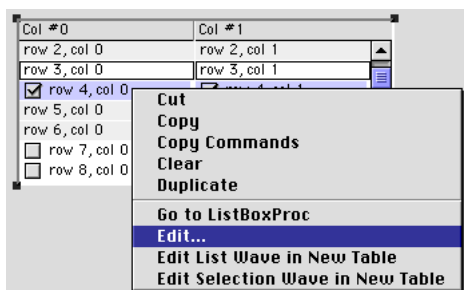
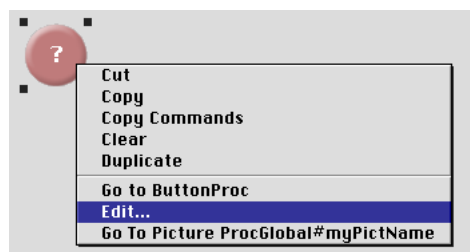


Note: You can also temporarily use the arrow tool without the toolbar showing by pressing Command-Option (*Macintosh*) or Ctrl+Alt (*Windows*). Holding down these keys while selecting from the Graph or Panel menu, you can also choose from the normally-disabled Add Controls and Select Control menus.

When you click a control with the arrow tool, small handles are drawn that allow you to resize the control. Note that some controls can not be resized in this way and some can only be resized in one dimension. You will know this when you try to resize a control and it doesn't budge. You can also use the arrow tool to reposition a control. You can select a control by name with the Select Control submenu in the Graph or Panel menu.

With the arrow tool, you can double-click most controls to get a dialog that modifies or duplicates the control. Charts and CustomControls do not have dialog support.

When you right-click (*Windows*) or Control-click (*Macintosh*) a control, you get a contextual menu that varies depending on the type of control.



You can select multiple controls, mix selections with draw objects and perform operations such as move, cut, copy, paste, delete with undo and align. You can't group buttons or use Send to Back as you can with Igor's draw objects.

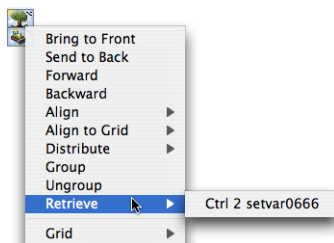
In panels, when you do a Select All it includes all controls and draw objects, but in the case of graphs, only draw objects are selected. This is because draw objects in graphs are used for presentation graphics whereas in panels they are used to construct the user interface.

If you want to copy controls from one window to another, simply use the Edit menu to copy and paste.

Note: If user controls are copied to the Clipboard, then the command and control names are also copied as text. This is handy when modifying controls that have no dialog support.

Hold down Option (*Macintosh*) or Alt (*Windows*) while choosing the Copy command to copy the complete set of procedures that create the copied controls.

If you copy a control from the extreme right side or bottom of a window, it may not be visible when you paste it into a smaller window. Use the smaller window's Retrieve submenu in the Mover menu to make it visible.



General Command Syntax

All of the control commands use the following general syntax:

```
ControlOperation Name [,keyword[=value] [,keyword[=value]]...]
```

Name is the control's name; it must be unique to the window containing the control. If *Name* is not already in use then a new control will be created. If a control with the same name already exists then that control will be modified, so that multiple commands using the same name result in only one control. This is useful for creating controls that require many keywords.

All keywords are optional. Not all controls accept all keywords, and some controls accept a keyword but do not actually use the value(s). The value for a keyword with one control can have a different form than the value for the same keyword used with other controls; the "value" keyword is a prime example of this. See the specific control operation documentation in Chapter V-1, **Igor Reference** for details.

Some controls utilize a format keyword to set a format string. The format string can be any printf style format that expects a single numeric value. Think of the output as being the result of the following command:

```
printf formatString , value_being_displayed
```

See the **printf** operation on page V-499 for a discussion of printf format strings. The maximum length of the format string is 63. The format is used only for controls that display numeric values and only for the principal value within a control (e.g., not used for the limit values for the ValDisplay control)

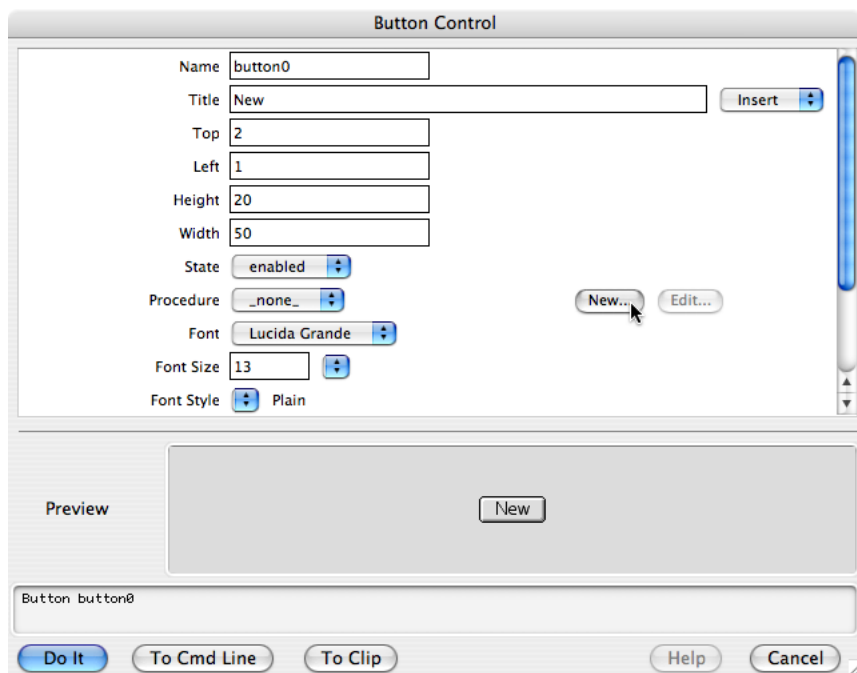
All of the clickable controls can optionally call a user-defined function usually when the user *releases* the mouse button. We use the term *action procedure* for such a function or macro. Each control passes one or more parameters to the action procedure. The dialogs for each control can create a blank user function with the correct parameters.

Read the following section on Buttons for general techniques that apply to all controls. You should also refer to Chapter V-1, **Igor Reference**, for each of the control-related operations and functions.

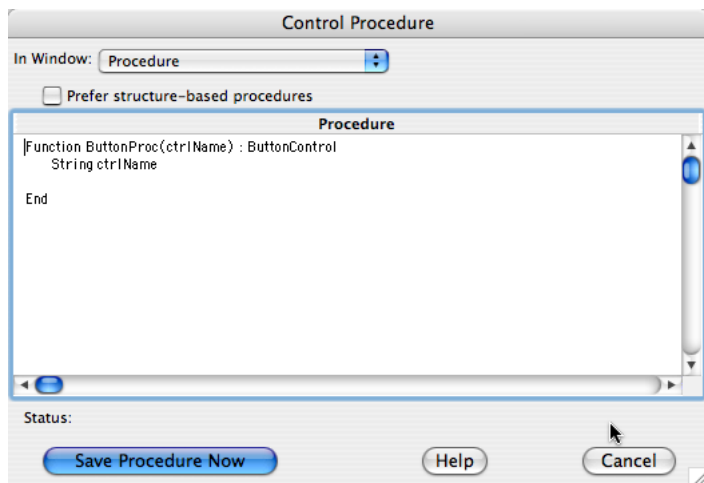
Button

The **Button** operation (page V-38) creates or modifies a rounded-edge or custom button with the title text centered in the button. The default font depends on the operating system, but you can change the font, font size, text color and use annotation-like escape codes (see **About Text Escape Codes** on page III-44). The amount of text does not change the button size, which you can set to what you want. However, the size of a custom button is determined only by the size of its Proc picture.

Here we create a simple button that will just emit a beep when pressed. Start by choosing the Add Button menu item in the Graph or Panel menu to get the Button Control dialog:



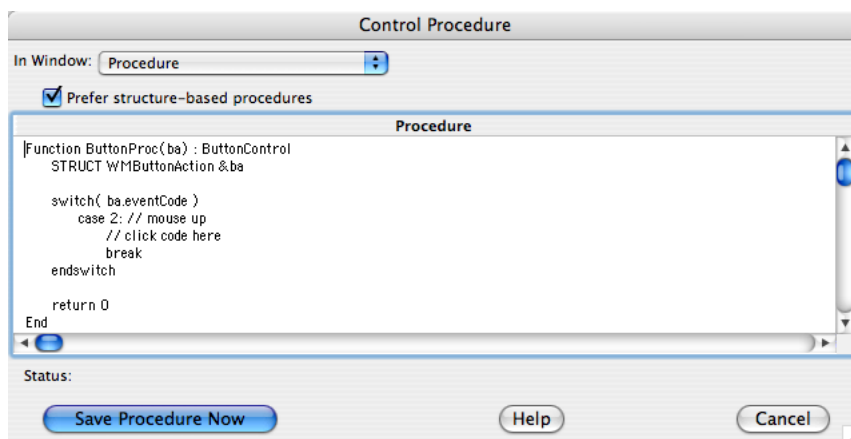
Clicking the procedure's New button brings up a dialog containing a procedure template that you can edit, rename, and save:



The controls can work with procedures using two formats: the “classic” procedure format used in Igor 3 and 4, and the “structure-based” format introduced in Igor 5.

Selecting the “Prefer structure-based procedures” checkbox creates new procedure templates using the structure-based format.

Tip: If you haven't edited a template (or if you delete the template), selecting or deselecting the checkbox will switch the template between the two formats:



Click Help to get help about the type of control the procedure works with. In this example, clicking Help would show the help file for the Button operation, which lists the details about the WMButtonAction structure in the Details section.

All we did in the above dialogs was click the New Procedure button, change the function name from ButtonProc to MyBeepProc and add the Beep command.

The fact that you can create the action procedure for a control in a dialog may lead you to believe that the procedure is stored with the button. This is not true. The procedure is actually stored in a procedure window. This way you can use the same action procedure for several controls. The parameters which are passed to a given procedure can be used to differentiate the individual controls.

As you can see from the above example, the user defined action procedure that you will need to write for buttons must have the following form:

```
Function ButtonProc(ctrlName) : ButtonControl
    String ctrlName
```

```
End
```

Replace ButtonProc with a descriptive name and fill in the body of the function. The ButtonControl subtype at the end of the function declaration line is optional but highly recommended. If it is present then the function will show up in the pop-up list of available procedures in the Button Control dialog. The other controls have similar subtypes.

It is legal for the action procedure to be written as a Macro (or Proc) rather than as a Function, but Functions are much faster than Macros.

You can use an additional kind of action procedure whose input parameter is a control-specific data structure. The Control Procedure dialog allows you to create either kind of action procedure. For more details, see **Control Structures** on page III-384, the **Button** operation (page V-38), and **Using Structures with Windows and Controls** on page IV-82.

Button Example

Here is how to make a button whose title alternates between Start and Stop.

Enter the following in the Procedure window:

```
Function StartStopButton(ctrlName) : ButtonControl
    String ctrlName

    if( cmpstr(ctrlName,"bStart") == 0 )
        Button $ctrlName,title="Stop",rename=bStop
        MyStartProc() // or whatever you want when start is pressed
    else
        Button $ctrlName,title="Start",rename=bStart
```

Chapter III-14 — Controls and Control Panels

```
MyStopProc() // or whatever you want when stop is pressed
endif
End
```

Additionally, you will also need to create the functions `MyStartProc()` and `MyStopProc()` to actually do something when the button is clicked.

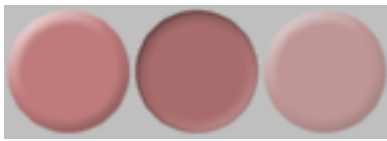
Then create the button in a graph or panel window with:

```
Button bStart, size={50,20}, proc=StartStopButton, title="Start"
```

Custom Button Example

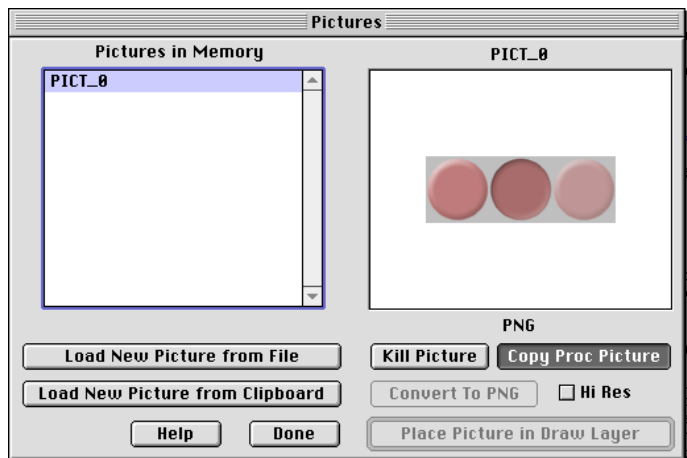
You can create custom buttons by following these steps:

- 1) Using a graphics-editing program, create a picture that shows the button in its normal (“relaxed”) state, then in the pressed-in state, and then in the disabled state. Each portion of the picture should be the same size:

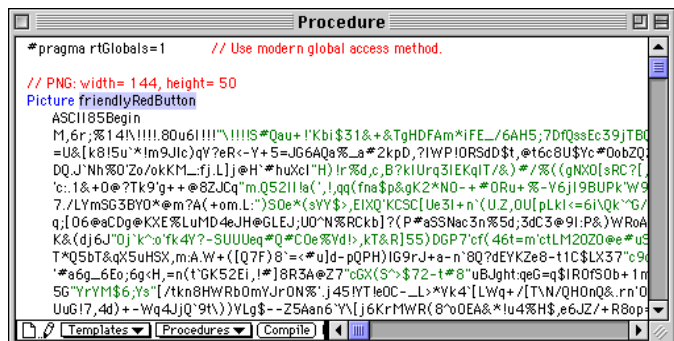


If the button blends into the background it will look better if the buttons are created on the background you will use in the panel. Igor looks at the pixels in the upper left corner, and if they are a light neutral color, Igor will omit those pixels when the button is drawn.

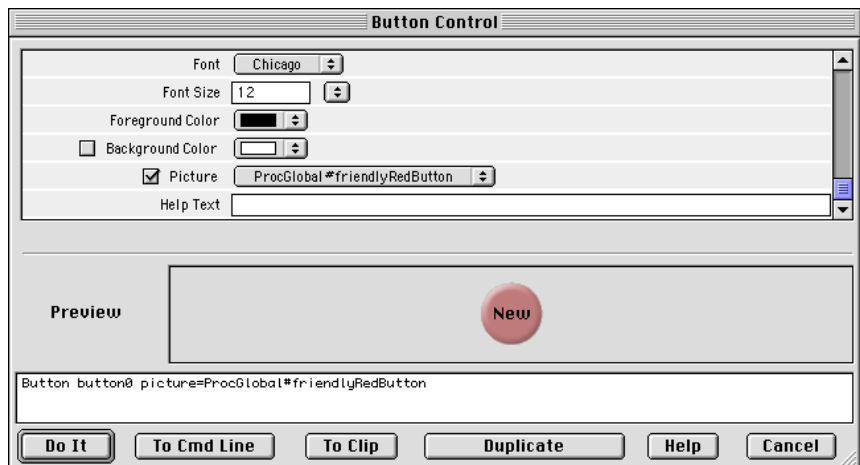
- 2) Copy the picture to the Clipboard.
- 3) Switch to Igor and open the Pictures dialog, and click Load New Picture from Clipboard.



- 4) Click Copy Proc Picture to create Proc Picture text on the Clipboard. Click Done.
- 5) Select a procedure window, paste the text, and give a suitable name to the picture:



6) Choose this picture in the Button Control dialog:

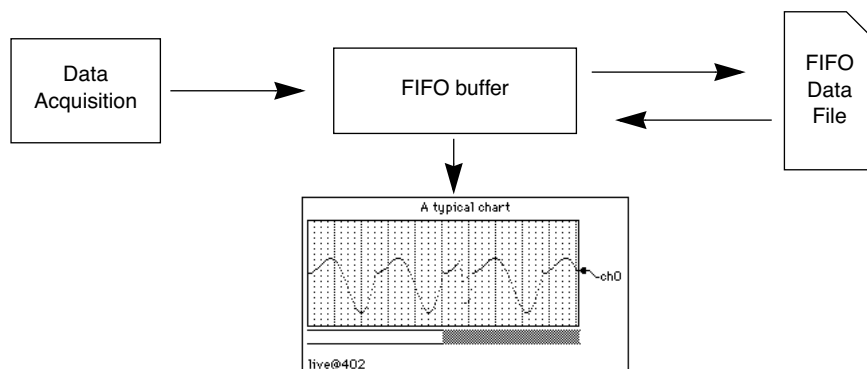


Charts and FIFOs

For further details see **FIFOs and Charts** on page IV-276.

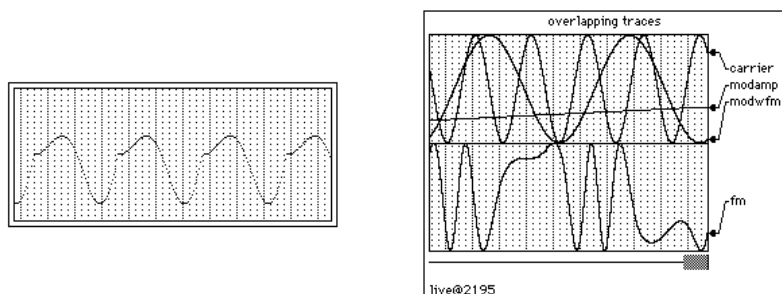
Chart controls can be used to emulate a mechanical chart recorder that writes on paper with moving pens as the paper scrolls by under the pens. Charts can be used to monitor data acquisition processes or to examine a long data record. Although programming a chart is quite involved, using a chart is very easy.

However, users of the Chart control do need to know the basics of the data acquisition process:



The First-In-First-Out (FIFO) buffer is an invisible Igor component that buffers the data coming from data acquisition hardware and software and also writes the data to a file. The data that is streaming through the FIFO can be observed using a Chart control. When data acquisition is finished the process can be reversed with data coming back out of the file and into the FIFO where it can be reviewed using the Chart. The FIFO file is optional but if missing then all data pushed out the end of the FIFO will be lost.

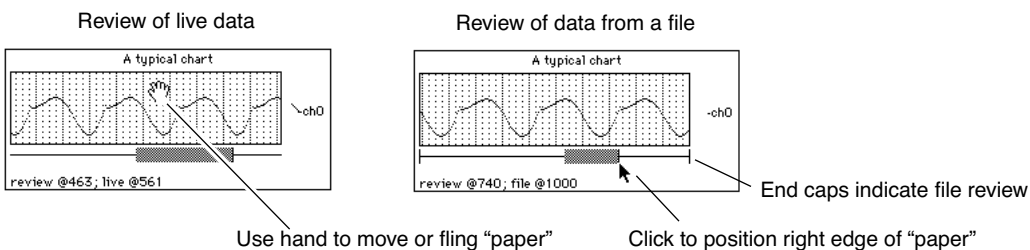
Chart controls can take on quite a number of forms from the simple to the sophisticated:



Charts can operate in two modes — live and review. When a chart is in live mode and data acquisition is in progress, the chart “paper” will scroll by from right to left under the influence of the acquisition process. When in review mode, you are in control of the chart. When you position the mouse over the chart area you will see that the cursor turns into a hand. You can move the chart paper right or left by dragging with the hand. If you give the paper a push it will continue scrolling until it hits the end. You can place the chart in review mode even as data acquisition is in progress by clicking in the paper with the hand cursor. To go back to live mode, give the paper a hard push to the left. When the paper hits the end then the chart will go to into live mode. You can also go back to live mode by clicking anywhere in the margins of the Chart.

Depending on the exact details of the data acquisition hardware and software you may run the risk of corrupting the data if you use review mode while acquisition is in progress. The person that created the hardware and software system you are using should have provided guidelines for the use of review mode during acquisition. In general, if the acquisition process is paced by hardware then it should be OK to use review mode.

In the above chart with lots of bells and whistles you may have noticed the line directly under the scrolling paper area. This line represents the current extent of data while the gray bar represents the data that is being shown in the chart. The right edge of the gray bar represents the right edge of the section of data being shown in the chart window. The above example is shown in live mode. Here are two examples shown in review mode:



While data acquisition is in progress, the horizontal line represents the extent of the data in the FIFO’s memory. After acquisition is over then the line includes all of the data in the FIFO’s output file (if any).

If you are in review mode while data acquisition is taking place you will notice that the gray bar will indicate the view area is moving even though the paper appears to be motionless. This is because the FIFO is moving out from under the chart. Eventually it will reach a position where the chart display can not be valid since the data it wants to display has been flushed off the end of the FIFO. When this happens the view area will go blank. Because it is very time-consuming for Igor to try to keep the chart updated in this situation your data acquisition rate may suffer.

CheckBox

The **CheckBox** operation (page V-46) creates or modifies a checkbox or a radio button. CheckBox controls automatically size themselves in both height and width. Checkboxes can optionally be connected to a global variable. For information on using checkboxes as radio buttons, see the example in the CheckBox reference section. You can use the font and fsize keywords to adjust the checkbox label.


The user-defined action procedure that you will need to write for CheckBoxes must have the following form:

```
Function CheckProc(ctrlName,checked) : CheckBoxControl
    String ctrlName
    Variable checked
```

End

The checked parameter will be set to the new checkbox value (0 or 1). Checkboxes do not usually need an action procedure since one can read the state of the checkbox with the **ControlInfo** operation (page V-63).

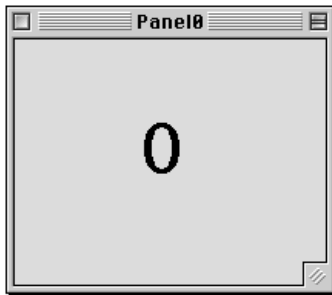
You can create custom checkboxes by following steps similar to those for custom Buttons (see **Custom Button Example** on page III-368), except the picture has six states side-by-side instead of three. The checkbox states are:

Image Order	Control State
Left	Deselected enabled.
	Deselected enabled and clicked down (about to be selected).
	Deselected disabled.
	Selected enabled.
	Selected enabled and clicked down (about to be deselected).
Right	Selected disabled.

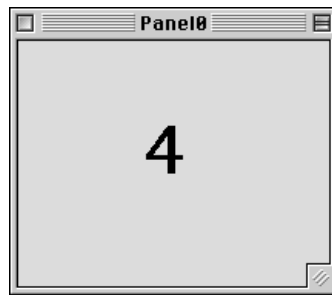
CustomControl

The CustomControl operation creates or modifies a custom control, all aspects of which are completely defined by the programmer. See the **CustomControl** operation on page V-93 for a complete description.

What you can create with a CustomControl can be fairly simple such as this counter that increments when you click on it.



Four clicks later:



The following code implements the counter custom control using the kCCE_frame event. In the panel, click on the number to increment the counter; also try clicking and then dragging outside the control.

```
static constant kCCE_mouseup= 2
static constant kCCE_frame= 12

// PNG: width= 280, height= 49
Picture Numbers0to9
ASCII85Begin
M,6r;%14!\!!!!.8Ou6I!!!$:!!!!R#Qau+!0#^OT5@]&TgHDFAm*iFE_/6AH5;7DfQssEc39jTBQ
=U"5QO:5u`*!m@2jnj"La,mA^'a?hQ[Z.[.,Kgd(1o5*(PSO8oS[GX%3u'11dT1)fII/"f-?Jq*no#
Qb>Y+UBKXKHQPQ&qYW88I,Ctm(`:C^]$4<ePf>Y(L\U!R2N7CEAn![N1I+[hTtr.VepqSG4R-;/+$3
IJE.V(>s0B@E@"n"ET+@5J9n_E:qeR_8:F1?m1=DM;mu.AEj!)]K4CUuCa4T=W)#(SE>uH[A4\;IG/
e]FqJ4u,2`*p=N5sc@qLD5bH89>gIBdF-1i6SF28oH@"3c2m)bDr&,UB$]i]/0bA.=qBR2#\ -D9E?O
2>3D>`($p(Kn)F8aF@)LYiXn[h2K):5@^kF?94)j*1Xtq1U2oFZmY.te?0G)EQ%5,RVT-c)DVa+%mP
%+bS*_hN$hC*8uCUuIWqTHJR.U?32`_B)(g_8e#*YXa>=faEdJsF]6iJlrQ@QAX7huJUmXj8:PBtb2
Y:DYf*Sci'Q"3_;@RDQA:A/([2s08r$hw)\B$XBGASJ:6OpC+GL<FjVfeNm20U<l<9J%cndX3'HP+k
R.Iv?U>ns*_;Zt[ ]6G6"Rb-*'Nm-E8]LXXXo7Ub>A**7Bm5cS*">HbQ&_RhmUe]$iu@T?Cci:e-`k
sE+H.GRSMT(9to;IZuH`T4%Yt<jF$+W?Yh6Q*`_C4sGig=L@DKoT%.H=#e_H"QEeeBVNTWBSMYr3dj
O=T%d&4kT9#CWPHS>kAG;3=or2(IK*IBF$^qK,+m0NSDK_!+e0#3fAI>HfKa<sk0641u\W@r+Y:$.i
```

Chapter III-14 — Controls and Control Panels

```

i$grCPR#&6, ;+>nTs_ IKS6XcYR)A$FjIC6Z_d2S!$R>_ZH+ [<p:JI0ub]\BhE(ORP@((KTRTGo;#SY
LT^9;D7X#km%UV20?$RS" FZoIF! (^FY-iL?n$%#o;-Wj (\PaBS6ZRQe@:kC>%ULrhTWNM=n@fUbRp
SKkLe\kJ)Sd]u7! ?pRJK- !XL[/MZX' "n4?a?JIK00k' KUm1IZ+roB=:Bq' $&E<#$Krp%p, E"4sI>[-
0F#^ff5SN':2fo) LNC?L4 (2ga=!aLm8) tVbGAM?L^l^=$D_YP7Z (sOFs) BL5er5G95p3?m%hM^lSr'
*E^O@8=u6hL`L$mPcq!B1-iHuGA6hiip%`cFj19>W?'E-&5T%Y.] i2A@1i%p8XJ5 [khh:&"JXYSC\r
10Ss8<Ye;S^"Nc0%-DFouAiPQ9Oemr!"sHH$JKt@!"d0E" 'M(P%: `p'15_10`!<nVt" TALQ>PF8WL
Z:#f!!!!j78?7R6=>B
ASCII85End
End

Structure CC_CounterInfo
  Int32 theCount // current frame of 10 frame sequence of numbers in
EndStructure

Function MyCC_CounterFunc(s)
  STRUCT WMCustomControlAction &s

  STRUCT CC_CounterInfo info

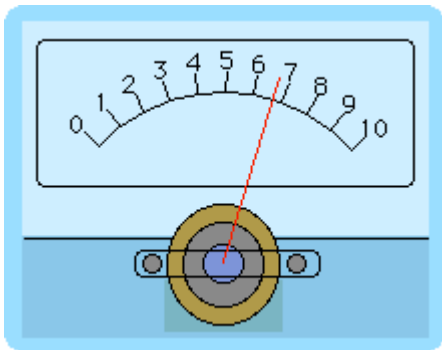
  if( s.eventCode==kCCE_frame )
    StructGet/S info,s.userdata
    s.curFrame= mod(info.theCount+(s.curFrame!=0),10)
  elseif( s.eventCode==kCCE_mouseup )
    StructGet/S info,s.userdata
    info.theCount= mod(info.theCount+1,10)
    StructPut/S info,s.userdata // will be written out to control
  endif

  return 0
End

Window Panel0() : Panel
  PauseUpdate; Silent 1 // building window...
  NewPanel /W=(69,93,271,252)
  CustomControl cc2,pos={82,46},proc=MyCC_CounterFunc,picture=
  {ProcGlobal#Numbers0to9,10}
EndMacro

```

You can create even more sophisticated controls, such as this voltage meter control.



The following code creates the voltage meter control. This example illustrates both draw and drawOSBM custom drawing along with several other events. Move the mouse over the surface of the meter to see how it responds.

```

static constant kCCE_mousemoved= 4
static constant kCCE_draw= 10
static constant kCCE_drawOSBM= 17

// PNG: width= 223, height= 180
Picture PanelMeterNoScale
ASCII85Begin
M,6r;%14!\!!!!.8Ou6I!!!!#V!!!#+#Qau+!1Tp) iW&rY&TgHDFAm*iFE_/6AH5;7DfQssEc39jTBQ
=U#(]?65u`*!mG0F:)b1m'nPpnh+J:S?W-P8e=<.L=]Rg.. (^<SC?kdiP^`-.-n1-1"nC6)E%8T9@)
rPq5J1UFnl#(^QpIATX4WfnXZ@%P$R:] i [\<HT3D#d/2%Xe=B,00rbf1e2OqDRDC@(\);D7jSgHeq>
\ (g_ 'S0C#?+**/o`kOVHs`5, FtA6/n>R1VFW&@%mZP0oV) $VJ, \0) `oG?Vc] h#!dQb1701G:P1*FA
XE-; ($H_-Zi35FAXE-; ($H_-ZfNp`qu=!9Y*7pjQc.fr?&0genMa4o8PI;D,Nr4fgui3g2rKUkP<5Q
1_K3EELh7caK>f [W"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Z
b@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'

```

```

=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=t+
U\u6:RW"Zb@'=t+U\u6:RW"Zb@'=q*,]R*nUT1h:1']eQWA9.FS:AD]ufh%imdAE,ZBjYKP.M+"[d\
'n,=D.ML$J<gcJhH>A0'\*Bl@^eTKJ)PC0'\<m+nBE'0)0#]f7R;%cFpb:=3G==4ioMiFiK<"dW,"
dkoPlB&r3(E\XPc)N@.U!Ttho2I0A9LJ%P6g"9,6U\5q=a_EhILrSA1;SE(X[+15[11+A\m&\V%C>O
VD5)@lp2,'I!>NN,Jg%iM!C!Pl89fMH^iJ@kEM`RKLvs0p,MPBo1V>Wj,@hNH,lJ.1hdrsu4dW,#dR
0Fb'5^-'01_R'pb)"`QUE`(\T\J*6D>: 7:=K$RdDke-i%0%ZST7kcfXt=\oc^=omAjPpMJ."J?I=
BMPBT9:cO[3HN!)4j3:C'r&rus*5M6dWfQb:Su,GP^'QihgbF1]oVW_k05Qof)GZh\&Q@fm+3&Oh&
aqJkV#Z:R@q8Z8+O^7prdjK'<X.O^V1_[3R$JES);g=^hPkMchR.j]X(IbK6*F#cN=&c)LJUNAQeI#
B&o9^fCtCHDk6n+:^l+?e`*:bVT"s;2]YuCl0RDOjGQ)MHWB,EB3CO%Mhg"CNA:.(g9I+E*ChNe.nV
&>lM;OK^,0iikucRjj'S1NPs'B?Br>S@JULjp"Wsu58CBQRpQ%XA5'<<bf40/fmbX1Od^a:b`gEgG6
Q$6<<n"%GkBXVqb[V+(UsA:mYG-G^[Y$!g`!@8i1%[RguG#n%J8;nB=>FH8n=JLQS:D$P`37/+4!S
\L36g0SCYDd78Y&tHC*sho#S%1A:lq:M`RO^`C#0q-LZ:.Eh:\pTV11096P-Zmm>-X5`--pB<0'#
$&E5sATscv%:cf/kLLpZsQUcrKD,'E>^#\CbN$WRIL[,RH-MC,qj09@'+YMP=@G%J*>,PmlTjOQ3g
!m/e$;!Kqj$H*k@;]o,bqaQ^GS"5$9,TGG)MEIJZI!tk]eT6M5XbPXT2/SNC9p'.9D@IY%hK#*VmFc
u/(3<7j?We3/?9<9.j[PI(ep2O^4+%^"1I2Plr%V%E3^V%,Ej,odd(g$F(eT*^pFi8)PlXN;QNVX,Am
gJBN(j2C92n)MZR)=k>cd-oF=Z7S/e$%BrSM^RMf^&0g:X^O@_&r3di3^L1B2IcB1,\*OpPP+I]X25
,HoJ/\*ne\g5KT5-kqG1X@X_p4Rir]<B(FtpM:lU%0pai.J%U9o^Q/gS@>MpF\U>anO.8Augk^@DBY
..%c.!3Vq#:R3Bi\'nZhX8Qh0iP*T)6GKc!&Wm_S0sGI%)mbpMe7Wr,p`4%G_\bg%,MJsR:gca80,
8CN4E=jKRdh[h*y(sL8pRm=U\$togFGpMY`k)W5(AF>.4_qYg2s-\tJ4%#[a]hi>m(KPp\7+i2qmVO
o3i'13W3n"JhmfB`?.90d@pI^Er:Ja;%foOpS^9frco"t0f>\ZR%nQm.Aq)q;@<OujVG:2CE*1jD1
-;2q\PC-1%`^$H'All@!*>4*.3ui&$ZtL/kn1<:QFTMfgXpDV;'=t+U)Dg%&&A1D@.Hc!!!!j78?7
R6=>B
ASCIi85End
End
Function drawneedle(v)
    Variable v // volts

    if( v<0 )
        v= 0
    elseif( v>10 )
        v= 10
    endif

    // Note: constants are specific to panel meter image
    Variable theta= 2.39 - 1.67*v/10
    Variable x0= 110, y0= 131,len= 96
    Variable x= x0 + len*cos(theta)
    Variable y= y0 - len*sin(theta)

    SetDrawEnv linefgc= (65535,0,0)
    DrawLine x0,y0,x,y
End

Function drawscale(vmin,vmax,n)
    Variable vmin,vmax,n

    variable i
    Variable theta0= 2.39 // Note: constants are specific to panel meter image
    Variable dtheta= -1.67/n
    Variable x00= 110, y00= 131,len= 85,ticklen=10,labelen=15
    String s

    SetDrawEnv textxjust= 1,textyjust= 1,save
    for(i=0;i<=n;i+=1)
        Variable theta= theta0 + i*dtheta
        Variable x0= x00 + len*cos(theta)
        Variable y0= y00 - len*sin(theta)
        sprintf s,"% .2g",vmin+i*(vmax-vmin)/n
        DrawLine x0,y0,x00 + (len+ticklen)*cos(theta),y00- (en+ticklen)*sin(theta)
        DrawText x00 + (len+labelen)*cos(theta),y00 - (len+labelen)*sin(theta),s
        if( i!=n )
            DrawLine x0,y0,x00 + len*cos(theta+dtheta),y00 - len*sin(theta+dtheta)
        endif
    endif
Endfor
End

Structure CC_MeterInfo
    double voltage // voltage value (0-10)
    STRUCT Point lastMouse
EndStructure

Function MyCC_MeterFunc(s)
    STRUCT WMCcustomControlAction &s

    STRUCT CC_MeterInfo info

```

Chapter III-14 — Controls and Control Panels

```
if( s.eventCode==kCCE_drawOSBM )
  drawscale(0,10,10)
elseif( s.eventCode==kCCE_draw )
  StructGet/S info,s.userdata
  drawneedle(info.voltage)
elseif( s.eventCode==kCCE_mousemoved )
  StructGet/S info,s.userdata
  // Beware: Next command is wrapped to fit on the page
  variable dist= sqrt((s.mouseLoc.h-info.lastMouse.h)^2+(s.mouseLoc.v-
info.lastMouse.v)^2)
  info.voltage= info.voltage + (dist-info.voltage)/10
  info.lastMouse= s.mouseLoc
  StructPut/S info,s.userdata // will be written out to control
  s.needAction= 1 // want redraw
endif

return 0
End

Window Panel0() : Panel
  PauseUpdate; Silent 1 // building window...
  NewPanel /W=(150,50,450,250)
  CustomControl cc3,pos={10,10},proc=MyCC_MeterFunc
  // This should be after the setting of the proc
  CustomControl cc3,picture= {ProcGlobal#PanelMeterNoScale,1}
EndMacro
```

GroupBox

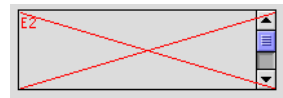
The GroupBox operation creates or modifies a listbox control. See the **GroupBox** operation on page V-238 for a complete description and examples.

ListBox

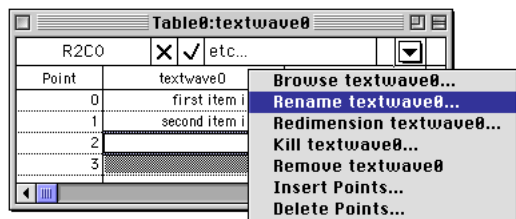
The ListBox operation creates or modifies a listbox control. See the **ListBox** operation on page V-336 for a complete description and examples.

The simplest listbox needs at least one text wave to contain the list items. Without the text wave, a listbox control has no list items.

```
ListBox list0 size={200,60},mode=1
```

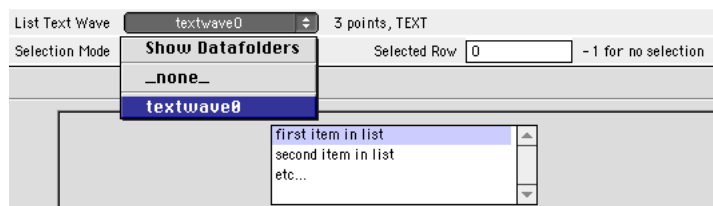


Create the text wave by opening a table and start typing a nonnumeric first list item (this is to make certain the wave is created as a text wave). You can rename the text wave by Control-clicking (*Macintosh*) or right-clicking (*Windows*) the name and choosing “Rename textwave0”.

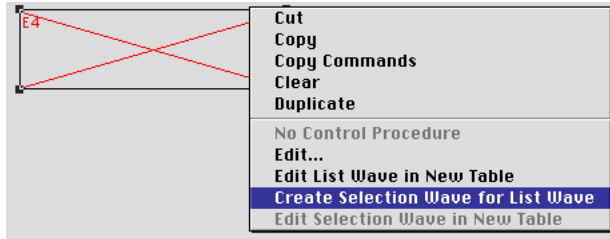


Note: If you want to create a text wave in a particular data folder, set the current data folder first (using the Data Browser in the Data menu).

Then select the wave you created in the ListBox Control dialog as the text wave for the list:



Right-clicking (*Windows*) or Control-clicking (*Macintosh*) a listbox shows a contextual menu for editing the list waves or action procedure, and to create a numeric selection wave (if the ListBox is a multiselection listbox):

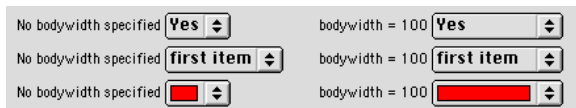


PopupMenu

The **PopupMenu** operation (page V-490) creates or modifies a pop-up menu control. Pop-up menus provide a choice of colors, line styles, patterns, markers, or text items:



The control automatically sizes itself as a function of the title or the currently selected menu item. You can specify the `bodyWidth` keyword to force the body (nontitle portion) of the pop-up menu to be a fixed size. You might do this to get a set of pop-up menus of nicely aligned with equal width. The `bodywidth` keyword also affects the nontext pop-up menus.



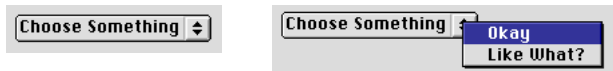
The `font` and `fsize` keywords affect only the title of a pop-up menu — the pop-up menu itself uses standard system fonts.

Unlike color, line style, pattern, or marker pop-up menus, text pop-up menu controls can operate in two distinct modes as set by the `mode` keyword's value.

If the argument to the `mode` keyword is nonzero then it is considered to be the number of the menu item to be the initial current item *and* displays the current item in the pop-up menu box. This is the *selector mode*. There is often no need for an action procedure since the value of the current item can be read at any time using the **ControlInfo** operation (page V-63).



If `mode` is zero then the title appears inside the pop-up menu box (hence the name *title-in-box mode*). This mode is generally used to select a command for the action procedure to execute. The current item has no meaning except when the pop-up menu is activated and the selected item (command) is passed to the action procedure.



The menu that pops up when the control is clicked is determined by a string expression that you pass as the argument to the `value` keyword.

Create the color, line style, pattern or marker pop-up menus by setting the string expression to one of these fixed values: `"*COLORPOP*"`, `"*LINESTYLEPOP*"`, `"*MARKERPOP*"`, or `"*PATTERNPOP*"`.

For text pop-up menus, the string expression must evaluate to a list of items separated by semicolons. For example:

```
PopupMenu name value= "Item 1; Item 2; Item 3"
PopupMenu name value= "_none_" + WaveList(";", ";", ";", ";")
```

It is possible to apply certain special effects to the menu items, such as disabling an item or marking an item with a check. See **Special Characters in Menu Item Strings** on page IV-114 for details.

Chapter III-14 — Controls and Control Panels

It is important to note that the literal text of the string expression is stored with the control rather than the results of the evaluation of the expression. The expression is reevaluated every time the user clicks on the pop-up menu box.

The string expression evaluates as if it were typed on the Command line within the current data folder. Additionally, it is important to specify the data folder containing any string variables, numeric variables, or waves used in the string expression:

```
PopupMenu name value=#"func(root:folder:wave0, root:gVar) "
```

For this reason, the expression can not use numeric or string variables that are local to a procedure since such variables cease to exist when the procedure finishes execution.

To incorporate the value of local variables in the value expression use the **Execute** operation:

```
String str= "\"_none_;first;second;\"" // str contains quotes
Execute "PopupMenu name value=" + str
```

The reason that the evaluation of the menu expression takes place when the user clicks on the menu is to ensure that dynamic menus such as the above WaveList example will reflect the *current* conditions rather than the conditions that were in effect when the PopupMenu control was *created*.

Because of this, the pop-up menu does not automatically update if the value of the string expression changes. You can use the **ControlUpdate** operation (page V-67) to force the pop-up menu to update. Here is an example:

```
NewPanel/N=PanelX
String/G gPopupMenu="First;Second;Third"
PopupMenu oneOfThree value=gPopupMenu // pop-up shows "First"
gPopupMenu="1;2;3" // pop-up is unchanged
ControlUpdate/W=PanelX oneOfThree // pop-up shows "1"
```

If the value expression can not be evaluated at the time the command is compiled, you can defer the evaluation of the expression by enclosing the value this way:

```
PopupMenu name value= #"pathToNonExistentGlobalString"
```

(the value=gPopupMenu example requires that gPopupMenu exist during compilation.)

If a deferred expression has quotes in it, they need to be “escaped” with backslashes (for a description of this syntax, see **When Dependencies are Updated** on page IV-206):

```
PopupMenu name value= #"\"_none_;" +UserFunc(\"foo\") "
```

The optional user defined action procedure is called after the user makes a selection from the popup menu. Popup menu procedures have the following form:

```
Function PopMenuProc(ctrlName, popNum, popStr) : PopupMenuControl
    String ctrlName
    Variable popNum
    String popStr
```

End

popNum will be the item number, *starting from one*, and popStr will be the text of the selected item. For the color pop-up menus the easiest way to determine the selected color is to use the **ControlInfo** operation (page V-63).

Another form of the action procedure uses structures. See **Using Structures with Windows and Controls** on page IV-82.

SetVariable

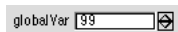
The **SetVariable** operation (page V-565) creates or modifies a SetVariable control. These controls are tied to numeric or string global variables (or even to one value in a numeric or text wave) and can be used to both view and set these values. When used with numeric variables, up/down arrows that the user can use to increment or decrement the variable will be drawn unless you set the increment value to zero (see the limits keyword).

You can set the width of the control but the height is determined from the font and font size. The width of the readout area is the width of the control less the width of the title and up/down arrows. However, you can use the `bodyWidth` keyword to specify a fixed width for the body (nontitle) portion of the control.

For example, executing the commands:

```
Variable/G globalVar=99
SetVariable setvar0 size={120,20},frame=1,font="Helvetica", value=globalVar
```

Results in the following SetVariable control:



To associate a SetVariable control with a variable that is not in the current data folder at the time SetVariable runs, you must use a data folder path:

```
Variable/G root:Packages:ImagePack:globalVar=99
SetVariable setvar0 value=root:Packages:ImagePack:globalVar
```

Unlike PopupMenu controls, SetVariable controls remember the current data folder when the SetVariable command executes. Thus an equivalent set of commands is:

```
SetDataFolder root:Packages:ImagePack
Variable/G globalVar=99
SetVariable setvar0 value=globalVar
```

Also see **Set Variable Controls and Data Folders** on page III-362.

You can control the style of the numeric readout via the `format` keyword. For example, the string `"%.2d"` will display the value with 2 digits past the decimal point. You should not use the format string to include text in the readout because Igor has to read back the numeric value. You may be able to add suffixes to the readout but prefixes will not work. When used with string variables the format string is not used.

The user defined action procedure that you may need to write for SetVariables must have the following form:

```
Function SetVarProc(ctrlName,varNum,varStr,varName) : SetVariableControl
    String ctrlName
    Variable varNum
    String varStr
    String varName
```

End

`varName` will be the name of the variable being used. If the variable is a string variable then `varStr` will contain its contents and `varNum` will be set to the results of an attempt to convert the string to a number. If the variable is numeric then `varNum` will contain its contents and `varStr` will be set to the results of a number to string conversion.

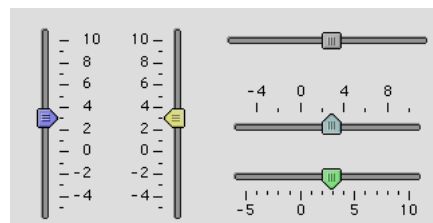
Note that when the user presses and holds in the up or down arrows then the value of the variable will be steadily changed by the increment value but your action procedure will not be called until the user releases the mouse button.

Another form of the action procedure uses structures. See **Using Structures with Windows and Controls** on page IV-82.

Slider

The **Slider** operation (page V-574) creates or modifies a slider control. The control can be used to set the value of a global variable or be used alone to indicate a value to be retrieved with the `ControlInfo` command. The value is changed by dragging the “thumb” part of the control. The slider can be drawn vertically or horizontally.

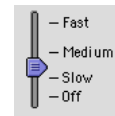
There are many options for labelling the numeric range such as setting the number of ticks.



Chapter III-14 — Controls and Control Panels

You can also provide custom labels in two waves (one numeric and another providing the corresponding text label):

```
Make/O tickNumbers= {0,25,60,100}
Make/O/T tickLabels= {"Off","Slow","Medium","Fast"}
Slider speed,pos={86,28},size={74,73}
Slider speed,limits={0,100,0},value= 40
Slider speed,userTicks={tickNumbers,tickLabels}
```

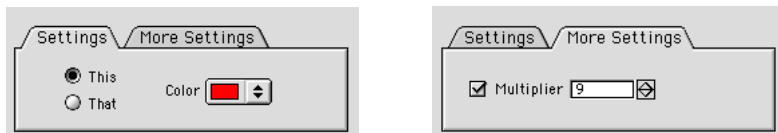


The action procedure for the slider can be used to apply the slider value to a process. The action procedure is called not only when the user moves the thumb, but also when the mouse button clicks down and up on the thumb, and when a procedure modifies the slider's controlled or controlling variable.

See the **Slider** operation on page V-574 for a complete description and more examples.

TabControl

The **TabControl** operation (page V-683) creates or modifies a TabControl control. Tabs are used to group controls into visible and hidden groups.



The tabs are numbered: the first tab is tab 0 and the second is tab 1, etc.

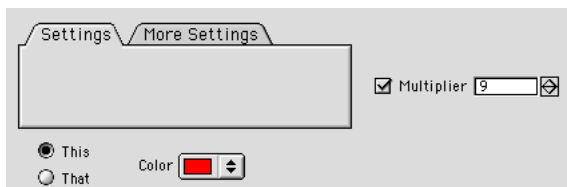
You add tabs to the control by providing additional tab titles:

```
TabControl tb, tabLabel(0)="Settings",tabLabel(1)="More Settings"
```

When you click on a tab, the control's action procedure receives the number of the clicked-on tab.

The showing and hiding of the controls are accomplished by user-written code in the tab control's action procedure. In this example, the "This", "That", and "Color" controls are shown when the "Settings" tab is clicked, and the "Multiplier" checkbox and SetVariable controls are hidden. When the "More Settings" tab is clicked, the action procedure makes opposite occur.

The simplest way to create a tabbed user interface is to create an over-sized panel with all the controls visible and not inside the tab control. Place controls in their approximate positions relative to one another:



By positioning the controls this way you can more easily modify each control until you are satisfied with them. Before you put them into the tab control, get a list of the nontab control names:

```
•Print ControlNameList("", "\r", "!tb") // all but "tb"
thisCheck
thatCheck
colorPop
multCheck
multVar
```

and figure out which tab you want them to be visible in:

Tab 0: Settings	Tab 1: More Settings
thisCheck	multCheck
thatCheck	multVar
colorPop	

Write the action procedure for the tab control to show and hide the controls:

```
Function TabProc(ctrlName,tabNum) : TabControl
    String ctrlName
    Variable tabNum

    Variable isTab0= tabNum==0
    Variable isTab1= tabNum==1

    // note: disable=0 means "show", disable=1 means "hide"
    ModifyControl thisCheck disable= !isTab0 // hide if not Tab 0
    ModifyControl thatCheck disable= !isTab0 // hide if not Tab 0
    ModifyControl colorPop disable= !isTab0 // hide if not Tab 0

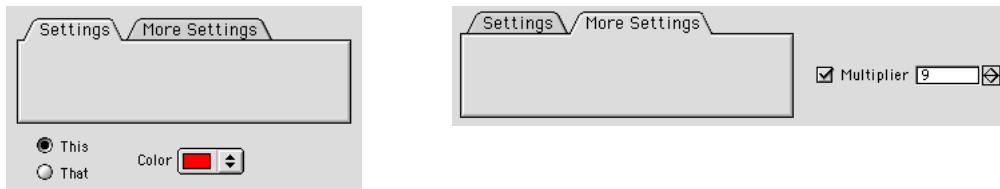
    ModifyControl multCheck disable= !isTab1 // hide if not Tab 1
    ModifyControl multVar disable= !isTab1 // hide if not Tab 1
    return 0
End
```

(A more elegant method, which will be important when you have many controls, is to systematically give the controls inside each tab a prefix or suffix that is unique to that tab, such as tab0_thisCheck, tab0_thatCheck, tab1_multVar, and use the ModifyControlList operation to show and hide the controls. See the **ModifyControlList** operation (page V-396) for an example.)

Then assign the action procedure to the tab control with the Tab Control dialog or a command like this:

```
TabControl tb proc=TabProc
```

Click on the tabs to see whether the showing and hiding is working correctly.



When it works correctly, click on one tab and then move the controls that belong inside into the tab area. Click on the next tab and then move that tab's controls into the tab area. You will need to use the operate mode to change the tab by clicking (and thus running the action procedure) and use the modify mode to move the controls. The "temporary selection" shortcut of pressing Command-Option (Macintosh) or Ctrl+Alt (Windows) is really handy here.

Save a recreation macro for the panel (Windows→Control→Window Control) to record the final control positions in a panel macro. Rewrite the macro as a function that initially creates the panel:

```
Function CreatePanel()
    DoWindow/K TabPanel // start over
    NewPanel/N=TabPanel/W=(596,59,874,175) as "Tab Demo Panel"
    TabControl tb,pos={15,19},size={250,80},proc=TabProc
    TabControl tb,tabLabel(0)="Settings"
    TabControl tb,tabLabel(1)="More Settings",value= 0
    CheckBox thisCheck,pos={53,52},size={39,14},title="This"
    CheckBox thisCheck,value= 1,mode=1
```

Chapter III-14 — Controls and Control Panels

```
CheckBox thatCheck, pos={53,72}, size={39,14}, title="That "  
CheckBox thatCheck, value= 0, mode=1  
PopupMenu colorPop, pos={126,60}, size={82,20}, title="Color "  
PopupMenu colorPop, mode=1, popColor= (65535,0,0)  
PopupMenu colorPop, value= #"\ "*COLORPOP*\ "  
CheckBox multCheck, pos={50,60}, size={16,14}, disable=1  
CheckBox multCheck, title="", value= 1  
SetVariable multVar, pos={69,60}, size={120,15}, disable=1  
SetVariable multVar, title="Multiplier", value=multiplier  
End
```

See the **TabControl** operation on page V-683 for a complete description and examples.

TitleBox

The **TitleBox** operation creates or modifies a **TitleBox** control. The control's text can be unchanging, or can be the contents of a global string variable. See the **TitleBox** operation on page V-705 for a complete description and examples.

ValDisplay

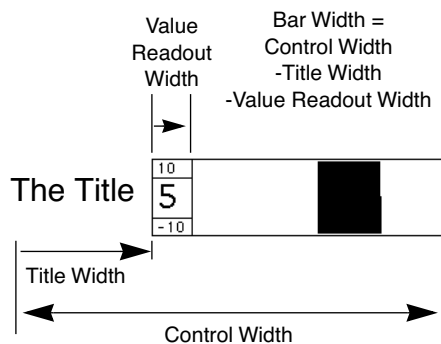
The **ValDisplay** operation (page V-715) creates or modifies a value display control. This is a very flexible and multifaceted control and can range from a simple numeric readout to a thermometer bar or a hybrid of both. A **ValDisplay** control is "connected" to a numeric expression that you provide as an argument to the value keyword. The display will be automatically updated whenever anything that the numeric expression depends on is changed.

ValDisplay controls evaluate their value expression in the context of the root data folder (see Chapter II-8, **Data Folders**, and **Programming with Data Folders** on page IV-148). To reference a data object that is not in the root, you must use a data folder path, such as "root:Folder1:var1".

Here are a few selected keywords extracted from the **ValDisplay** operation on page V-715:

```
size={width,height}  
barmisc={lts, valwidth}  
limits={low,high,base}
```

The appearance of the **ValDisplay** control depends primarily on the *valwidth* and *size* parameters and the width of the title. However, you can use the *bodyWidth* keyword to specify a fixed width for the body (non-title) portion of the control. Essentially, space for each element is allocated from left to right, with the title receiving first priority. If the control width hasn't all been used by the title, then the value readout width is the smaller of *valwidth* pixels of room or what is left. If the control width hasn't been used up, the bar is displayed in the remaining control width:



Here are the various major possible forms of **ValDisplay** controls. Some of these examples modify previous examples; check the names of the **ValDisplay** controls. For instance, the second bar-only example is a modification of the *valdisp1* control created by the first bar-only example.

Numeric Readout Only

```
// default readout width (1000) is >= default control width (50)
ValDisplay valdisp0 value=K0
```

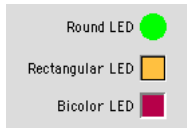


LED Display

```
// Create the three LED types.
ValDisplay led1,pos={67,17},size={75,20},title="Round LED"
ValDisplay led1,limits={-50,100,0},barmisc={0,0},mode=1
ValDisplay led1,bodyWidth= 20,value= #"K1",zeroColor=(0,65535,0)

ValDisplay led2,pos={38,48},size={104,20},title="Rectangular LED"
ValDisplay led2,frame=5,limits={0,100,0},barmisc={0,0},mode=2
ValDisplay led2,bodyWidth= 20,value= #"K2"
ValDisplay led2,zeroColor= (65535,49157,16385)

ValDisplay led3,pos={60,76},size={82,20},title="Bicolor LED"
ValDisplay led3,limits={-40,100,-100},barmisc={0,0},mode= 2
ValDisplay led3,bodyWidth= 20,value= #"K3"
```



Bar Only

```
// readout width = 0
ValDisplay valdisp1,frame=1,barmisc={12,0},limits={-10,10,0},value=K0
K0= 5 // halfway from base of 0 to high limit of 10.
```

The nice thing about a bar-only ValDisplay is that you can make it 5 to 200 pixels tall whereas with a numeric readout, the height is set by the font sizes of the readout and printed limits.

```
// Set control height= 80
ValDisplay valdisp1, size={50,80}
```



Numeric Readout and Bar

```
// 0 < readout width (50) < control width (150)
ValDisplay valdisp2 size={150,20},frame=1,limits={-10,10,0}
ValDisplay valdisp2 barmisc={0,50},value=K0 // no limits shown
```



Optional Limits

Whenever the numeric readout is visible, the optional limit values may be displayed too.

```
// Set limits font size to 10 points. Readout widths unchanged.
ValDisplay valdisp2 barmisc={10,50}
ValDisplay valdisp0 barmisc={10,1000}
```



Chapter III-14 — Controls and Control Panels

Optional Title

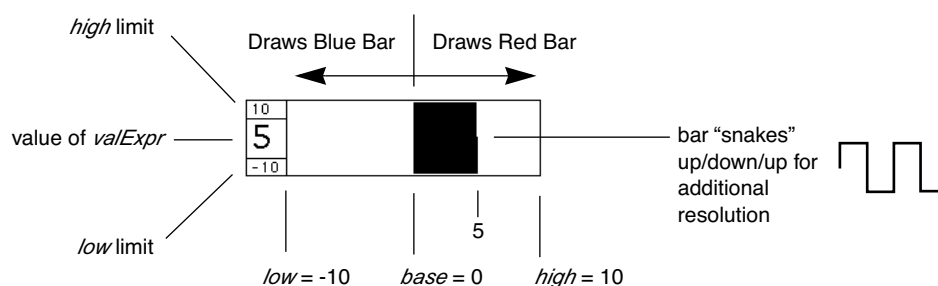
The control title steals horizontal space from the numeric readout and the bar, pushing them to the right. You may need to increase the control width to prevent them from disappearing.

```
// Add titles. Readout widths, control widths unchanged.
ValDisplay valdisp2 title="Readout+Bar"
ValDisplay valdisp0 title="K0="
```



The limits values *low*, *high*, and *base* and the value of *valExpr* control how the bar, if any, is drawn. The bar is drawn from a starting position corresponding to the *base* value to an ending position determined by the value of *valExpr*, *low* and *high*. *low* corresponds to the left side of the bar, and *high* corresponds to the right. The position that corresponds to the *base* value is linearly interpolated between *low* and *high*.

For example, with *low* = -10, *high*=10, and *base*= 0, a *valExpr* value of 5 will draw from the center of the bar area (0 is centered between -10 and 10) to the right, halfway from the center to the right of the bar area (5 is halfway from 0 to 10):



You can force the control to not draw bars with fractional parts by specifying *mode*=3.

Killing a Control

You can kill (delete) a control from within a procedure using the **KillControl** operation (page V-321). This might be useful in creating control panels that change their appearance depending on other settings.

You can interactively kill a control by selecting it with the arrow tool or the Select Control submenu and press Delete. Moving a control out of the window does not delete it; you just end up with a control that is offscreen.

Getting Information About a Control

You can use the **ControlInfo** operation (page V-63) to obtain information about a given control. This is useful to obtain the current state of a checkbox or the current setting of a pop-up menu.

Updating a Control

You can use the **ControlUpdate** operation (page V-67) to cause a given control to redraw with its current value. You can use this in a user-defined function or macro after changing the value or appearance of a control and to display the changes before the normal graph or panel update occurs.

Help Text for User-Defined Controls

You can easily add help Igor Tips (*Macintosh*) or context-sensitive/status line help text (*Windows*) for your controls. Each dialog has a button titled Igor Tips (*Macintosh*) or Edit Help (*Windows*) that leads to a subdi-

alog where you can edit the help text. You can also use the command line. You are limited to using a total of 255 characters for your help message. Here is an example:

```
Button button0 title="Beep", help={"This button beeps."}
```

This creates a button with a Macintosh Igor Tip, or under Windows, this text appears as context-sensitive help and in the status line when the mouse passes over the control. The help text that appears on the status line is limited to the first 127 characters of text or the text up to the first line break. The `help={"This button beeps."}` section sets the help text for the button. You can use the help keyword with the Check-box, PopupMenu, ValDisplay, and SetVariable operations as well as the Button operation.

There is no way to specify help for the individual items in a user-defined pop-up menu.

Modifying a Control

The control operations create a new control if the name parameter doesn't match a control already in the window. The operations modify an existing control if the name does match a control in the window, but generate an error if the control kind doesn't match the operation.

For example, if a panel already has Button control named `button0`, you can modify, say, the disable state for that button with another `Button button0` command:

```
Button button0 disable=1 // hide
```

However, if you use a Checkbox instead of Button, you will get a "button0 is not a Checkbox" error.

You can use the **ModifyControl** operation (page V-395) and **ModifyControlList** operation (page V-396) to modify a control without needing to know what kind of control it is:

```
ModifyControl button0 disable=1 // hide
```

This is really handy when used in conjunction with tab controls.

Disabling and Hiding a Control

All controls support the keyword "disable=*d*" where *d* can be 0 (normal operation), 1 (hidden), or 2 (user input disabled). Charts and ValDisplays do not change appearance when `disable=2` because they are read-only.

SetVariables also have the `noedit` keyword. This appears to be redundant with the `disable=2` mode but there is one quasi-important difference: `noedit` still allows user input via the up or down arrows but `disable=2` does not.

Background Color

The background color of Panel windows (see **Control Panels** on page III-389) and the area at the top of a graph as reserved by the **ControlBar** operation (page V-63) is a shade of gray chosen to match the standard Macintosh or Windows system look. This gray is used when the background color is the default pure white where the red, green and blue components are all 65535. Any other color, including not quite pure white, will be honored.

However some controls or portions of controls are drawn by the system and may look out of place if you choose a different background color. The `cbRGB` keyword is used to set the control background in Panels (**ModifyPanel** operation on page V-419) and Graphs (**ModifyGraph (colors)** operation on page V-415).

Most controls are drawn with a background color that matches the window background color giving the impression of transparency. A few controls are drawn with true transparent labels and no background color is needed.

For special purposes, you can specify a background color for an individual control using the `labelBack` keyword (see the reference descriptions of the individual controls for further details).

Chapter III-14 — Controls and Control Panels

On the Macintosh prior to Igor Pro 4, the background color of controls was white. This matched the window color which was also white. The new technique is somewhat nonbackwards compatible and a few people may prefer the old white on white look. In many cases simply setting the window background to slightly off-white:

```
ModifyGraph cbRGB=(65534,65534,65534)
```

will do the job. In other cases, you may need to specify the background color of individual controls using the `labelBack` keyword.

On Windows, the background color of the controls themselves is dependent on the current Appearances Settings. Specifically, the background color of controls is set by the 3D Objects color in the Appearance Tab of the Display Properties control panel. You should be aware that any color you choose now may not result in pleasing esthetics when these settings are changed in the future.

On Windows if you do not explicitly set the background color, the standard Windows 3D Objects color is used. This works well with buttons and other standard Windows controls, even when the user changes the 3D Objects color later. You can change the background color to track the 3D Objects color by choosing the current 3D Objects color from the color pop-up palette, or by executing a command to set the background to maximum Macintosh white:

```
ModifyGraph cbRGB=(65535,65535,65535)
```

```
ModifyPanel cbRGB=(65535,65535,65535)
```

Control Structures

The action procedure for a control can also use a predefined, built-in structure as a parameter to the function. The control will use this more efficient method whenever the function properly matches the structure prototype for a control, otherwise it will use the “old-style” method.

An action procedure using a structure has the format:

```
Function newActionProcName(CB_Struct)
    STRUCT WMCheckboxAction &CB_Struct
    ...
End
```

The names of the various control structures are listed in the next table. You should consult the reference information for each individual control to see what members each control structure contains.

Control Type	Structure Name
Button	WMButtonAction
CheckBox	WMCheckboxAction
CustomControl	WMCustomControlAction
ListBox	WMListboxAction
PopupMenu	WMPopupAction
SetVariable	WMSetVariableAction
Slider	WMSliderAction
TabControl	WMTabControlAction

Action functions should respond only to documented `eventCode` values. Other event codes may be added along with more fields. Although the return value is not currently used, action functions should always return zero.

The constants used to specify the size of structure char arrays are internal to Igor Pro and may change.

Control Structure Example

This example illustrates the extended event codes available with a Button control (as well as a application of user data). The function prints various text messages to the History area depending what actions you take while in the button area.

```
Function structureTest()
    NewPanel
    Button b0,proc= NewButtonProc
End

Structure MyButtonInfo
    Int32 mousedown
    Int32 isLeft
EndStructure

Function NewButtonProc(s)
    STRUCT WMBUTTONACTION &s

    STRUCT MyButtonInfo bi
    Variable biChanged= 0

    StructGet/S bi,s.userdata
    if( s.eventCode==1 )
        bi.mousedown= 1
        bi.isLeft= s.mouseLoc.h < (s.ctrlRect.left+s.ctrlRect.right)/2
        biChanged= 1
    elseif( s.eventCode==2 || s.eventCode==3 )
        bi.mousedown= 0
        biChanged= 1
    elseif( s.eventCode==5 )
        print "Enter button"
    elseif( s.eventCode==6 )
        print "Leave button"
    endif

    if( s.eventCode==4 )                // mousemoved
        if( bi.mousedown )
            if( bi.isLeft )
                printf "L"
            else
                printf "R"
            endif
        else
            printf "*"
        endif
    endif
    if( biChanged )
        StructPut/S bi,s.userdata      // written out to control
    endif

    return 0
End
```

Control Structure eventMod Field

The eventMod field appears in the built-in structure for each type of control. It is a bitfield defined as follows:

EventMod Bit	Meaning
Bit 0	Left mouse button is down.
Bit 1	Shift key is down.
Bit 2	Option (Macintosh) or Alt (Windows) is down.

EventMod Bit	Meaning
Bit 3	Command (Macintosh) or Ctrl (Windows) is down.
Bit 4	Contextual menu click: right-click or Control-click (Macintosh).

See **Setting Bit Parameters** on page IV-12 for details about bit settings.

Control Structure `blockReentry` Field

The `blockReentry` field appears in the built-in structure for each type of control. It allows you to prevent Igor from sending your control action procedure another event while you are servicing the first event. This is useful for action procedures that take a long time to service a click event. In such cases you typically do not want to service a second click until you finish servicing the first. This technique prevents an accidental double-click on a button from invoking a time-consuming procedure twice.

You tell Igor that you want to block further events until your action procedure returns by setting the `blockReentry` field to 1 when your action procedure is called:

```
Function ButtonProc(ba) : ButtonControl
    STRUCT WMBUTTONACTION &ba

    // Tell Igor not to invoke ButtonProc again until this invocation is finished
    ba.blockReentry = 1
    . . .
```

Igor tests this field before invoking your action procedure while it is already running from a previous invocation. You do not need to test this field or reset it to 0 - just set it to 1 to block reentry.

Control Structure `blockReentry` Advanced Example

This example further illustrates the use of the `blockReentry` field. It is of interest only to those who want to experiment with this issue.

The `ReentryDemoPanel` procedure below creates a panel with two buttons. Each button prints a message in the history area when the action procedure receives the "mouse up" message, then pauses for two seconds, and then prints another message in the history before returning. The pause is a stand-in for a procedure that takes a long time.

The top button does not block reentry so, if you click it twice in quick succession, the action procedure is reentered and you get nested messages in the history area.

The bottom button does block reentry so, if you click it twice in quick succession, the action procedure is not reentered.

Because of architectural differences, reentry of the action procedure occurs on Macintosh but not on Windows so on Windows, both buttons behave the same.

```
Function ButtonProc(ba) : ButtonControl
    STRUCT WMBUTTONACTION &ba

    switch( ba.eventCode )
        case 2: // mouse up
            // Block bottom button only
            ba.blockReentry= CmpStr(ba.ctrlName,"Block") == 0
            print "Start button ",ba.ctrlName
            Variable t0= ticks
            do
                DoUpdate
            while(ticks < (t0+120) )
            Print "Finish button",ba.ctrlName
            break
```

```

    endswitch

    return 0
End

Window ReentryDemoPanel() : Panel
    PauseUpdate; Silent 1// building window...
    NewPanel /K=1 /W=(322,55,622,255)
    Button NoBlock,pos={25,10},size={150,20},proc=ButtonProc,title="No Block Reentry"
    Button Block,pos={25,50},size={150,20},proc=ButtonProc,title="Block
Reentry"
End

```

User Data for Controls

You can store arbitrary data with a control using the `userdata` keyword. You can set user data for the following controls: `Button`, `CheckBox`, `CustomControl`, `ListBox`, `PopupMenu`, `SetVariable`, `Slider`, and `TabControl`.

Each control has a primary, unnamed user data that is used by default. You can also store an unlimited number of different user data strings by specifying a name for each different user data string. The name can be anything you desire as long as it is a legal Igor name.

You can retrieve information from the default user data using the **ControlInfo** operation (page V-63), which returns such information in the `S_UserData` string variable. To retrieve any named user data, you must use the **GetUserData** operation (page V-224).

Although there is no size limit to how much user data you can store, it does have to be generated as part of the recreation macro for the window when experiments are saved. Consequently, huge user data can slow down experiment saving and loading.

User data is intended to replace or reduce the usage of global variables.

Control User Data Examples

A simple example of user data with a button:

```

NewPanel
Button b0,userdata="user data for button b0"
Print GetUserData("", "b0", "")

```

A more complex example using user data for buttons. Copy the following code into the Procedure window of a new experiment and run the `Panel()` macro. Then click on the buttons.

```

Structure mystruct
    Int32 nclicks
    double lastTime
EndStructure

Function ButtonProc(ctrlName) : ButtonControl
    String ctrlName

    STRUCT mystruct s1
    String s= GetUserData("", ctrlName, "")
    if( strlen(s) == 0 )
        print "first click"
    else
        StructGet/S s1,s
        // Warning: Next command is wrapped to fit on the page.
        printf "button %s clicked %d time(s), last click = %s\r",ctrlName, s1.nclicks,
Secs2Date(s1.lastTime, 1 )+" "+Secs2Time(s1.lastTime,1)
    endif
    s1.nclicks += 1
    s1.lastTime= datetime
    StructPut/S s1,s
    Button $ctrlName,userdata= s
End

Window Panel0() : Panel
    PauseUpdate; Silent 1 // building window...
    NewPanel /W=(150,50,493,133)

```

```

SetDrawLayer UserBack
Button b0,pos={12,8},size={50,20},proc=ButtonProc,title="Click"
Button b1,pos={65,8},size={50,20},proc=ButtonProc,title="Click"
Button b2,pos={119,8},size={50,20},proc=ButtonProc,title="Click"
Button b3,pos={172,8},size={50,20},proc=ButtonProc,title="Click"
Button b4,pos={226,8},size={50,20},proc=ButtonProc,title="Click"
EndMacro

```

Action Procedures for Multiple Controls

You can use the same action procedure for different controls of the same type (for all the Buttons in one window, for example). The name of the control is passed to the action procedure so that it can know which control was clicked. This is usually the name of the control *in the target/active window*, which is what most control operations assume.

When you call an action procedure from another procedure (rather than because a control was clicked), the name of the control may not be sufficient to identify the control; the window name may also be needed. This must be passed in through a global string. The window name can then be used to specify which window the named control is in by adding `win=winName` to the command, as in:

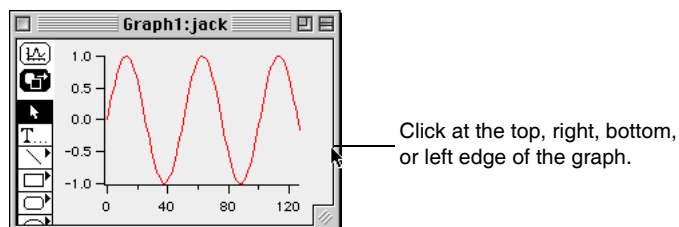
```
Button button0 win=$graphNameStr, title="Hello"
```

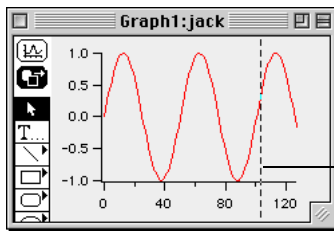
Controls in Graphs

Although controls can be placed anywhere in a graph, you can and should reserve an area just for controls at the edge of a graph window. Controls in graphs operate much more smoothly if they reside in these reserved areas. The **ControlBar** operation (page V-63) or the Control Bar dialog can be used to set the height of a nonembedded control area at the top of the graph.

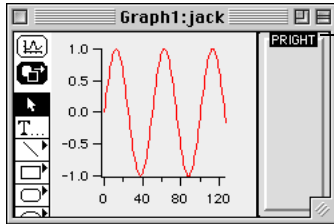


You can also embed a panel in the graph, which gives you more freedom on where to place the controls (see Chapter III-4, **Embedding and Subwindows** for further details and **Embedding into Control Panels** on page III-390 for a different approach). The simplest way to add a panel is to click near the edge of the graph and drag out a control area:

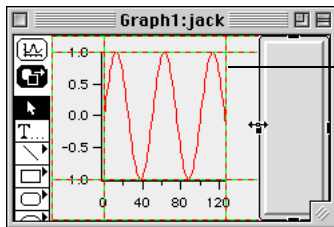




Drag the dashed line to define the inside edge of the embedded panel.



PRIGHT is the name of the resulting embedded panel subwindow. The label disappears in "operate" mode.



Adjust the position of the embedded window by clicking the subwindow frame and dragging its handles. The dashed lines represent the edges of the plot and graph areas, and the subwindow frame snap and attach to them.

The background color of a control area or embedded panel can be set by clicking the background to exit any subwindow layout mode, and then Control-clicking (*Macintosh*) or right-clicking (*Windows*) in the background and then selecting a color from the contextual menu's pop-up color palette. See **Background Color** on page III-383 for further details.

The contextual menu adjusts the style of the frame around the panel.

You can use the same contextual menu to remove an embedded panel, leaving only the bare control area underneath. Remove the control area by dragging the inside edge back to the outside edge of the graph.

Drawing Limitations

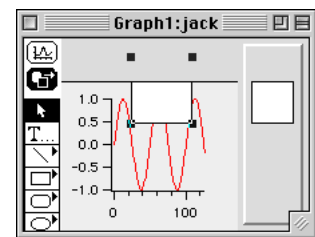
The drawing tools can not be used in bare control areas of a graph. If you want to create a fancy set of controls with drawing tools, you will have to embed a panel subwindow into the graph.

Updating Problems

You may occasionally run into certain updating problems when you use controls in graphs. One class of update problems occurs when the action procedure for one control changes a variable used by a ValDisplay control in the same graph and also forces the graph to update while the action procedure is being executed. This short-circuits the normal chain of events and results in the ValDisplay not being updated.

You can force the ValDisplay to update using the **ControlUpdate** operation (page V-67). Another solution is to use a control panel instead of a graph.

The ControlUpdate operation can also solve problems in updating pop-up menus. This is described above under **PopupMenu** on page III-375.



Control Panels

Control panels are windows designed to contain controls. The **NewPanel** operation (page V-440) is used to create a panel.

Chapter III-14 — Controls and Control Panels

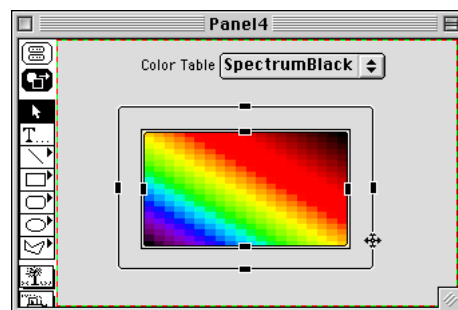
Drawing tools can be used in panel windows to decorate control panels. Only two drawing layers are provided and both are behind the controls layer. There is probably little reason to distinguish between the user and prog layers since users have no reason to draw in panels.

The Drawing tools can be used in panel windows to decorate control panels. Background color can be set by Control-clicking (*Macintosh*) or right-clicking (*Windows*) in the background and then selecting a color from the pop-up color palette. See **Background Color** on page III-383 for further details.

Embedding into Control Panels

You can embed a graph, table, notebook, or another panel into a control panel window (see Chapter III-4, **Embedding and Sub-windows** for further details). You may find this preferable to putting control areas around a graph when you have many controls, or you may embed a graph in order to display an annotation, colorscale, or image plot. Use the contextual menu while in drawing mode to add an embedded window. Click on the frame of the embedded window to adjust the size and position.

You can use a notebook subwindow in a control panel to display status information or to accept lengthy user input. See **Notebooks as Subwindows in Control Panels** on page III-94 for details.



Exterior Subwindows

Exterior subwindows are panels that act like subwindows but live in their own windows attached to a host graph window. The host graph and its exterior subwindows move together and, in general, act as single window. Exterior subwindows have the advantage of not disturbing the host graph and, unlike normal subwindows, are not limited in size by the host graph.

Note: Exterior subwindows must be panels and the only host supported is a graph window.

To create an exterior subwindow panel, use **NewPanel** with the /EXT flag in combination with /HOST.

Floating Panels

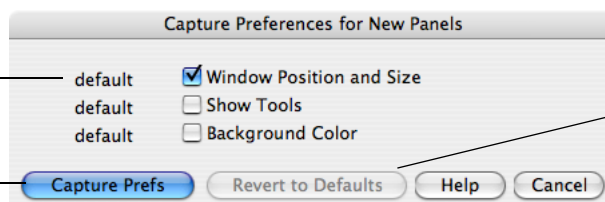
Floating control panels float above all other windows (except dialogs). To create a floating panel, use **NewPanel** with the /FLT flag.

Control Panel Preferences

Control panel preferences allow you to control what happens when you create a new control panel. To set preferences, create a panel and set it up to your taste. We call this your *prototype* panel. Then choose Capture Panel Prefs from the Panel menu.

Indicates that the current window position and size are the factory defaults.

Captures preferences for the selected items from the active control panel window.



Resets preferences for the selected items to the factory defaults.

Preferences are normally in effect only for *manual* operations, not for automatic operations from Igor procedures. This is discussed in more detail in Chapter III-17, **Preferences**.

When you initially install Igor, all preferences are set to the factory defaults. The dialog indicates which preferences you have changed.

The preferences affect the creation of new panels only.

Selecting the Show Tools category checkbox captures (or reverts) whether or not the drawing tools palette is initially shown or hidden when a new panel is created.

Controls Shortcuts

Action	Shortcut (<i>Macintosh</i>)	Shortcut (<i>Windows</i>)
To show or hide a panel's or graph's tool palette	Press Command-T.	Press Ctrl+T.
To move or resize a user-defined control without using the tool palette	Press Command-Option and click the control. With Command-Option still pressed, drag or resize it.	Press Ctrl+Alt and click the control. With Ctrl+Alt still pressed, drag or resize it.
To add a user-defined control without using the tool palette	Press Command-Option and choose from the Panel or Graph menu's Add Control submenu.	Press Ctrl+Alt and choose from the Panel or Graph menu's Add Control submenu.
To modify a user-defined control	Press Command-Option and double-click the control. This displays a dialog for modifying all aspects of the control. If the control is already selected, you don't need to press Command-Option.	Press Ctrl+Alt and double-click the control. This displays a dialog for modifying all aspects of the control. If the control is already selected, you don't need to press Ctrl+Alt.
To edit a user-defined control's action procedure	With the panel in modify mode (tools showing, second icon from the top selected) press the Control key and click the control. This displays a contextual menu with a "Go to <action procedure>" item.	With the panel in modify mode (tools showing, second icon from the top selected) right-click the control. This displays a contextual menu with a "Go to <action procedure>" item.
To create an embedded graph or table in the panel	With the panel in modify mode, press the Control key and click the panel background. Choose the subwindow type from the resulting contextual menu's "New" submenu.	With the panel in modify mode, right-click the panel background. Choose the subwindow type from the resulting contextual menu's "New" submenu.
To change an embedded window's border style	With the panel in modify mode, press the Control key and click the embedded window. Choose the border style from the resulting contextual menu's "Frame" and "Style" submenus.	With the panel in modify mode, right-click the embedded window. Choose the border style from the resulting contextual menu's "Frame" and "Style" submenus.
To remove an embedded window	With the panel in modify mode, press the Control key and click the embedded window. Choose the Delete from the resulting contextual menu.	With the panel in modify mode, right-click the embedded window. Choose the Delete from the resulting contextual menu.
To eliminate a control area at the edge of a graph	In modify mode or while pressing Command-Option, click the inside edge of the control area and drag it to the outside edge of the graph.	In modify mode or while pressing Ctrl+Alt, click the inside edge of the control area and drag it to the outside edge of the graph.
To nudge a user-defined control's position	Select the control and press arrow keys. Press Shift to nudge faster.	Select the control and press arrow keys. Press Shift to nudge faster.