Chapter
III-14

Controls and Control Panels

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.

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 PopupMenu, 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.

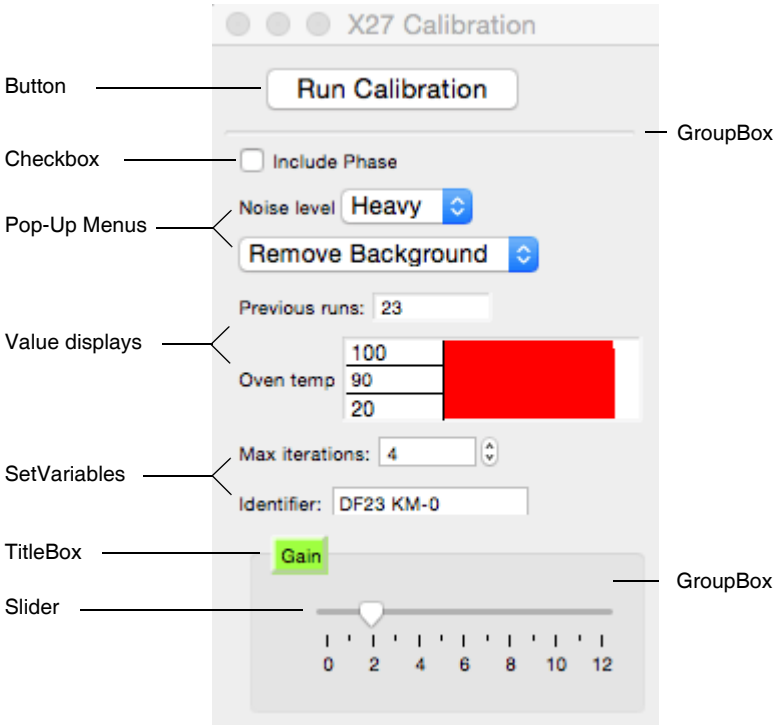
When the top icon is selected or when the icons are hidden, you are in the *use* or *operate* mode. You can momentarily switch to the *modify* or *draw* mode by pressing Command-Option (*Macintosh*) or Ctrl+Alt (*Windows*). Use this to drag or resize a control as well as to double-click it. Double-clicking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

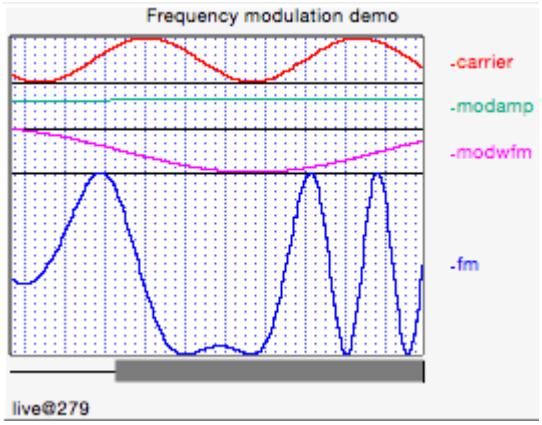
If nothing happens when you click a button, then there is no procedure assigned to the button. If the procedure window(s) haven't been compiled, clicking a button that has an assigned procedure will produce an error dialog.

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.

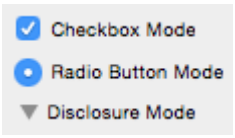


For further discussion of using chart controls, see **Using Chart Recorder Controls** on page IV-296.

Although programming a chart is quite involved, using a chart is actually very easy. See **FIFOs and Charts** on page IV-291 for details.

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.

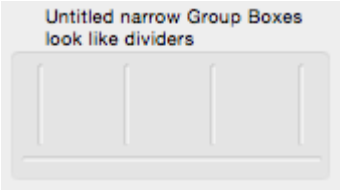


CustomControl

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 **Creating Custom Controls** on page III-377 for examples.

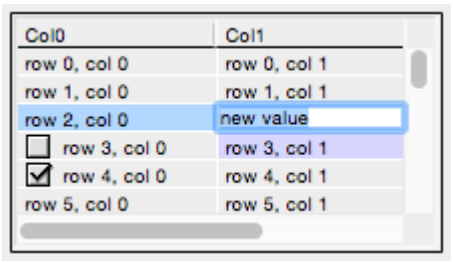
GroupBox

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.

SetVariable

SetVariable 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, SetVariable 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. SetVariable controls can also be used with global string variables to display or set short one line strings. SetVariable 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 SetVariable control may run a procedure if the programmer has specified one.

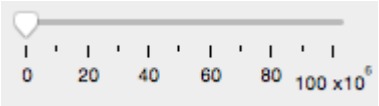
SetVariable Controls and Data Folders

SetVariable 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 **SetVariable** on page III-370.

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.*

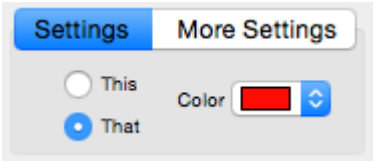
Sliders

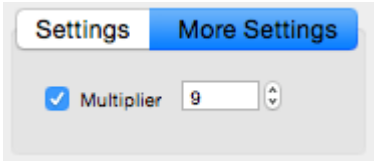
Slider controls can be used to graphically select either discrete or continuous values. When used to select discrete values, a slider is similar to a pop-up menu or a set of radio buttons. Sliders can be live, updating a variable or running a procedure as the user drags the slider, or they can be configured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



TabControl

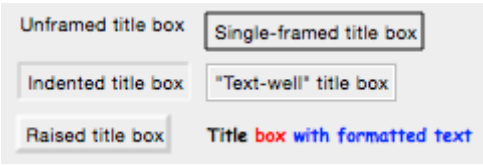
TabControls are used to create complex panels containing many more controls than would otherwise fit. When the user clicks on a tab, the programmers procedure runs and hides the previous set of controls while showing the new set.





TitleBox

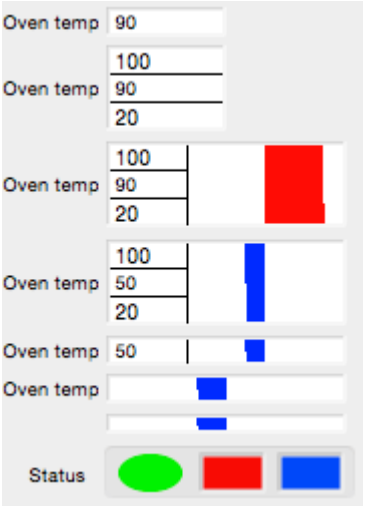
TitleBox controls are mainly decorative elements. They are used provide explanatory text in a control panel. They may also be used to display textual results. The text can be unchanging, or can be the contents of a global string variable. In either case, the user can't inadvertently change the text.



ValDisplays

ValDisplay controls display numeric or string values in a variety of forms ranging from a simple numeric readout to a thermometer bar. Regardless of the form, ValDisplays 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 ValDisplay 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 procedure can create buttons and checkboxes, but creating charts and custom controls requires more expertise. Most controls can be created and modified using dialogs that you invoke via the Add Controls submenu in the Graph or Panel menu.

The Add Controls and Select Control menus are enabled only when the arrow tool in the tool palette is selected. To do this, choose Show Tools from the Graph or Panel menu and then click the second icon from the top in the graph or panel tool palette.

You can temporarily use the arrow tool without the tool palette showing by pressing Command-Option (*Macintosh*) or Ctrl+Alt (*Windows*). While you press these keys, the normally-disabled Add Controls and Select Control submenus are enabled.

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 drawing objects, and perform operations such as move, cut, copy, paste, delete, and align. These operations are undoable. You can't group buttons or use Send to Back as you can with drawing objects.

In panels, when you do a Select All, the selectin includes all controls and drawing objects, but in the case of graphs, only drawing objects are selected. This is because drawing 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. You can also duplicate controls using copy and paste.

When you copy controls to the clipboard, the command and control names are also copied as text. This is handy for programming.

Press Option (*Macintosh*) or Alt (*Windows*) while choosing Edit-Copy to copy the complete commands for creating 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 tool palette icon 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 among controls in the window containing the control. If *Name* is not already in use then a new control is created. If a control with the same name already exists then that control is 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. The value for a keyword with one control can have a different form from the value for the same keyword used with a different type of control. 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
```

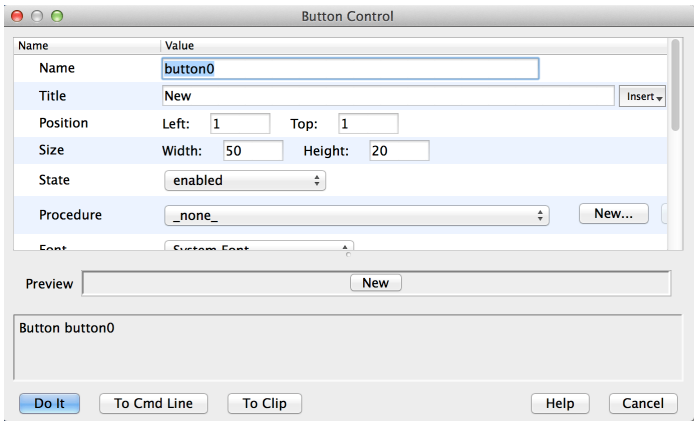
See the **printf** operation on page V-653 for a discussion of printf format strings. The maximum length of the format string is 63 bytes. The format is used only for controls that display numeric values.

All of the clickable controls can optionally call a user-defined function when the user releases the mouse button. We use the term *action procedure* for such a function. 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.

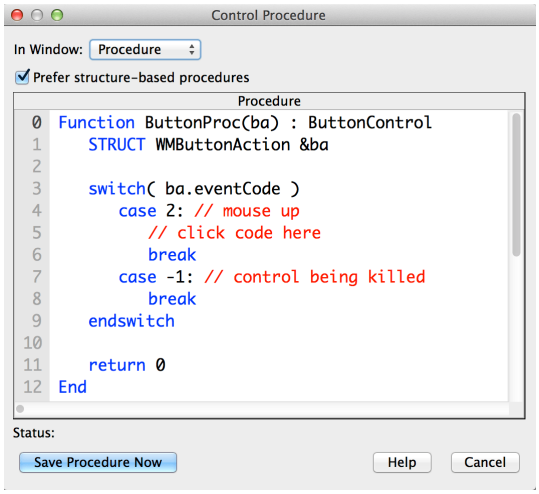
Creating Button Controls

The **Button** operation (page V-45) 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 **Annotation Escape Codes** on page III-53). The amount of text does not change the button size, which you can set to what you want.

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 invoke 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. Here we started with the standard ButtonControl template, replaced the default name with `MyBeepProc`, and added the `Beep` command:



The controls can work with procedures using two formats: the old 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.

The "Prefer structure-based procedures" checkbox is checked by default because structure-based procedures are recommended for all new code. If you uncheck this checkbox before editing the template, Igor switches the template to the old procedure format. Use of the old format is discouraged.

Click Help to get help about the Button operation. In the Details section of the help, you can find information about the WMButtonAction structure.

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.

For more information on using action procedures, see **Control Structures** on page III-388, the **Button** operation (page V-45), and **Using Structures with Windows and Controls** on page IV-95.

Button Example

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

Enter the following in the procedure window:

```
Function MyStartProc()
   Print "Start"
End

Function MyStopProc()
   Print "Stop"
End

Function StartStopButton(ba) : ButtonControl
   STRUCT WMButtonAction &ba

   switch(ba.eventCode)
      case 2:                          // Mouse up
         if (CmpStr(ba.ctrlName,"bStart") == 0)
            Button $ba.ctrlName,title="Stop",rename=bStop
            MyStartProc()
         else
```

```
            Button $ba.ctrlName,title="Start",rename=bStart
            MyStopProc()
         endif
         break
   endswitch

   return 0
End
```

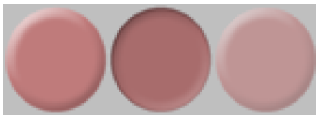
Now execute:

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

Custom Button Control Example

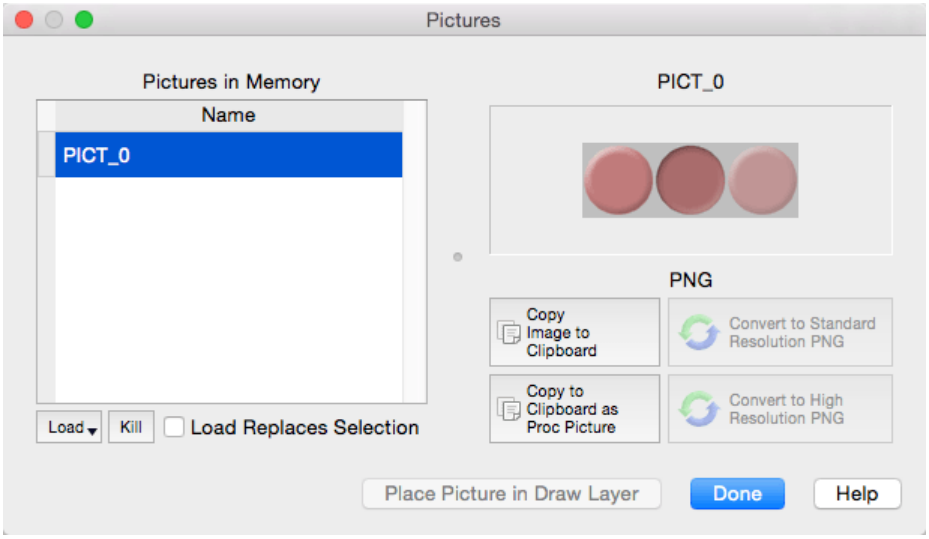
You can create custom buttons by following these steps:

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 omits those pixels when the button is drawn.

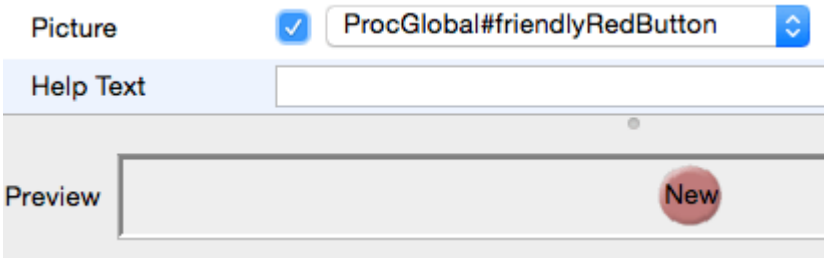
2. Copy the picture to the clipboard.
3. Switch to Igor,choose Misc→Pictures, click Load and then From Clipboard.



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

```
// PNG: width= 144, height= 50
Picture friendlyRedButtor
    ASCII85Begin
    M,6r;%14!\!!!!.8Ou6I!!!"\!!!!S#Qau+!'Kbi$31&+&TgHDFAm*:
    =U&[k8!5u`*!m9JIc)qV?eR<-Y+5=JG6AQa%_a#2kpD,?lWP!0RSdD:
    DQ.J`Nh%O'Zo/okKM_:fj.L]j@H`#huXcI"H)!r%d,c,B?klUrq3IEI
    'c:.1&+O@?Tk9'g++@8ZJCq"m.Q52ll!a(',!,qq(fna$p&gK2*N0--
```

7. Activate an existing control panel or create a new one.
8. If the tool palette is not showing, choose Panel→Show Tools.
9. Choose Panel-Add Control→Add Button to display the Button Control dialog.
10. Locate the Picture setting in the dialog, check the checkbox, and select the proc picture from the pop-up menu:



11. Locate the Size setting in the dialog and set the appropriate size for the button:

Creating Chart Controls

The **Chart** operation creates or modifies a chart control. There is no dialog support for chart controls. You need at least intermediate-level Igor programming skills to create a functional chart control.

For further information, see **FIFOs and Charts** on page IV-291.

Creating Checkbox Controls

The **CheckBox** creates or modifies a checkbox or a radio button.

CheckBox controls automatically size themselves in both height and width. They can optionally be connected to a global variable.

For an example of using checkbox controls as radio buttons, see the reference documentation for the **CheckBox** operation.

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

```
Function CheckProc(cba) : CheckBoxControl
   STRUCT WMCheckboxAction &cba

   switch(cba.eventCode)
      case 2:               // Mouse up
         Variable checked = cba.checked
         break
      case -1:              // Control being killed
         break
   endswitch

   return 0
End
```

The checked structure member is set to the new checkbox value:; 0 or 1.

You often do not need an action procedure for a checkbox because you can read the state of the checkbox with the **ControlInfo** operation.

You can create custom checkboxes by following steps similar to those for custom buttons (see **Custom Button Control Example** on page III-375), except that the picture has six states side-by-side instead of three. The checkbox states are:

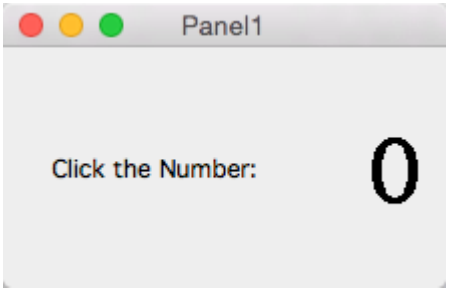
Image Order	Control State
Left	Deselected enabled.
	Deselected enabled and clicked down (about to be selected).
	Deselected disabled.
	Selected enabled.
	Selected enabled and clicked down (about to be deselected).
Right	Selected disabled.

Creating Custom Controls

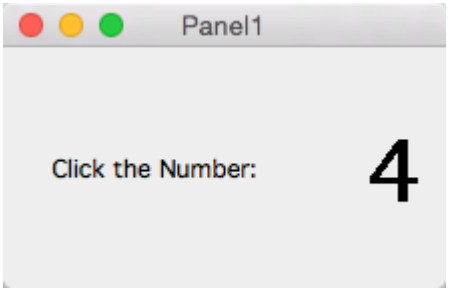
The CustomControl operation creates or modifies a custom control, all aspects of which are completely defined by the programmer. See the **CustomControl** operation on page V-114 for a complete description.

The examples in this section are also available in the Custom Controls demo experiment. Choose File→Example Experiments→Feature Demos 2→Custom Control Demo.

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:

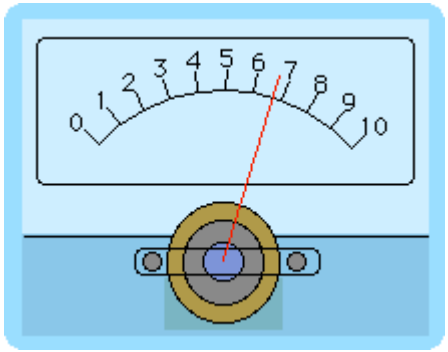


The following code implements the counter custom control using the `kCCE_frame` event. In the panel, click on the number to increment the counter; also try clicking and then dragging outside the control.

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

```
// PNG: width= 280, height= 49
Picture Numbers0to9
    ASCII85Begin
    M,6r;%14!\!!!!.8Ou6I!!!$:!!!!R#Qau+!00#^OT5@]&TgHDFAm*iFE_/6AH5;7DfQssEc39jTBQ
    =U"5QO:5u`*!m@2jnj"La,mA^'a?hQ[Z.[.,Kgd(1o5*(PSO8oS[GX%3u'11dTl)fII/"f-?Jq*no#
    Qb>Y+UBKXKHQpQ&qYW88I,Ctm(`:C^]$4<ePf>Y(L\U!R2N7CEAn![N1I+[hTtr.VepqSG4R-;/+$3
    IJE.V(>s0B@E@"n"ET+@5J9n_E:qeR_8:Fl?m1=DM;mu.AEj!)]K4CUuCa4T=W)#(SE>uH[A4\;IG/
    e]FqJ4u,2`*p=N5sc@qLD5bH89>gIBdF-1i6SF28oH@"3c2m)bDr&,UB$]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*8uCJuIWqTHJR.U?32`_B)(g_8e#*YXa>=faEdJsF]6iJlrQ@QAX7huJUmXj8:PBTb2
    Y:DYf*Sci'Q"3_;@RDQA:A/([2sO8r$hW)\B$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
    i$grCPR#&6,;+>nTs_IKS6XcYR)A$fJiC6Z_d2S!$R>_ZH+[<p:JI0ub]\BhE(0RP@((KTRTGo;#SY
    LT^9;D7X#km%UV20?$RS"FZoIF!(`FY-iL?n$%#o;-Wj(\PaBS6ZRQe@:kC>%ULrhTWLNM=n@fUbRp
    SKkLe\kJ)Sd]u7!?pRJk-!XL[/MZX'"n4?a?JIKO0k'KUm1IZ+roB=:Bq'$&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$mPcq!Bl-iHuGA6hiip%`cFjl9>W?'E-&5T%Y.]i2A@1i%p8XJ5[khb:&"JXYSC\r
    10Ss8<Ye;S^"Nc0%-DFouAiPQ9OemnR!"sHH$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.



Choose File→Example Experiments→Feature Demos 2→Custom Control Demo to try this control and see the code that implements it.

Creating GroupBox Controls

The GroupBox operation creates or modifies a listbox control. See the **GroupBox** operation on page V-289 for a complete description and examples.

Creating ListBox Controls

The **ListBox** operation creates or modifies a listbox control.

We will illustrate listbox creation by example.

1. Create a panel with a listbox control:

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

   The simplest functional listbox needs at least one text wave to contain the list items. Without the text wave, a listbox control has no list items. In this state, the listbox is drawn with a red X over the control.

2. We need a text wave to contain the list items:

   ```
   Make/O/T textWave0 = {"first list item", "second list item", "etc..."}
   ```

3. Choose Panel→Show Tools.

   This puts the panel in edit mode so you can modify controls.

4. Double-click the listbox control to invoke the ListBox Control dialog.
5. For the List Text Wave property, select the wave you created to assign it as the list's text wave.
6. Click Do It.

   You now have a functional listbox control.

7. Click the operate (top) icon, or choose Panel→Hide Tools, so you can use, rather than edit, the list.

In this example, we created a single-selection list. You can query the selection by calling **ControlInfo** and checking the V_Value output variable.

See the **ListBox** for a complete description and further examples.

Right-clicking (*Windows*) or Control-clicking (*Macintosh*) a listbox shows a contextual menu with commands for editing the list waves and action procedure, and for creating a numeric selection wave, if the control is a multi-selection listbox.

Creating PopupMenu Controls

The **PopupMenu** creates or modifies a pop-up menu control. Pop-up menus are usually used to provide a choice of text items but can also present colors, line styles, patterns, and markers.

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 (non-title 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 non-text 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-74).

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 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. For example:

```
PopupMenu name value="Item 1;Item 2;Item 3;"
```

To create the color, line style, pattern or marker pop-up menus, set the string expression to one of these fixed values:

```
"*COLORPOP*"
"*LINESTYLEPOP*"
"*MARKERPOP*"
"*PATTERNPOP*"
```

For text pop-up menus, the string expression must evaluate to a list of items separated by semicolons. This can be a fixed literal string or a dynamically-calculated string. 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-125 for details.

The literal text of the string expression is stored with the control rather than the results of the evaluation of the expression. Igor evaluates the expression when the PopupMenu value=<value> command runs and reevaluates it every time the user clicks on the pop-up menu box. This reevaluation ensures that dynamic menus, such as created by the WaveList example above, reflect the conditions at click time rather than the conditions that were in effect when the PopupMenu control was created.

When the user clicks and Igor reevaluates the string expression, the procedure that created the pop-up menu is no longer running. Consequently, its local variables no longer exist, so the string expression can not reference them. To incorporate the value of local variables in the value expression use the **Execute** operation:

```
String str = <code that generates item list>    // str is a local variable
Execute "PopupMenu name value=" + str
```

Igor evaluates the string expression as if it were typed on the command line. You can not know what the current data folder will be when the user clicks the pop-up menu. Consequently, if you want to refer to objects in specific data folders, you must use full paths. For example:

```
PopupMenu name value=#"func(root:DF234:wave0, root:gVar)"
```

Because of click-time reevaluation, the pop-up menu does not automatically update if the value of the string expression changes. Normally this is not a problem, but you can use the **ControlUpdate** operation (page V-79) to force the pop-up menu to update. Here is an example:

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

If the string 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:

In some cases, the string expression can not be compiled at the time the PopupMenu command is compiled because it references a global object that does not yet exist. In this case, you can prevent a compile-time error by using this special syntax:

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

If a deferred expression has quotes in it, they need to be escaped with backslashes:

```
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(pa) : PopupMenuControl
   STRUCT WMPopupAction pa

   switch(pa.eventCode)
      case 2:        // Mouse up
         Variable popNum = pa.popNum      // 1-based item number
         String popStr = pa.popStr        // Text of selected item
         break
      case -1:       // Control being killed
         break
   endswitch
End
```

`pa.popNum` is the item number, *starting from one*, and pa.`popStr` is 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**.

Creating SetVariable Controls

The **SetVariable** operation (page V-729) creates or modifies a SetVariable control. SetVariable controls are useful for both viewing and setting values.

SetVariable controls are tied to numeric or string global variables, to a single element of a wave, or to an internal value stored in the control itself. To minimize clutter, you should use internal values in most cases.

When used with numeric variables, Igor draws up and down arrows that the user can use to increment or decrement the value.

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
```

creates 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 **SetVariable Controls and Data Folders** on page III-370.

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.

Often it is sufficient to query the value using **ControlInfo** and you there is no need for an action procedure. If you want to do something every time the value is changed, then you need to create an action procedure of the following form:

```
Function SetVarProc(sva) : SetVariableControl
   STRUCT WMSetVariableAction sva

   switch(sva.eventCode)
      case 1:                        // Mouse up
      case 2:                        // Enter key
      case 3:                        // Live update
         Variable dval = sva.dval
         String sval = sva.sval
         break
      break
      case -1:                       // Control being killed
         break
   endswitch

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.

If the value is a string, then sva.sval contains the value. If it is numeric, then sva.dval contains the value. sva.isStr is 0 for numeric values and non-zero for string values.

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.

Creating Slider Controls

The **Slider** creates or modifies a slider control.

A slider control is tied to a numeric global variables or to a numeric internal value stored in the control itself. To minimize clutter, you should use internal values in most cases. The value is changed by dragging the "thumb" part of the control.

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

You can also provide custom labels in two waves, one numeric and another providing the corresponding text label. For example:

```
NewPanel
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}
```

Often it is sufficient to query the value using **ControlInfo** and you there is no need for an action procedure. If you want to do something every time the value is changed, then you need to create an action procedure.

Igor calls the action procedure when the user drags the thumb, when the user clicks the thumb, and when a procedure modifies the slider's global variable, if any.

See the **Slider** operation on page V-745 for a complete description and more examples.

Creating TabControl Controls

The **TabControl** 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, the second is tab 1, etc.

A default tab control has one tab:

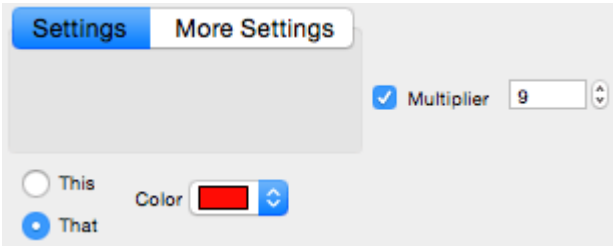
```
NewPanel/W=(150,50,650,400)
TabControl tb, tabLabel(0)="Settings", size={400,250}
```

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

```
TabControl tb, 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 your action procedure. In this example, the This, That, and Color controls are shown when the Settings tab is clicked, and the Multiplier checkbox is hidden. When the More Settings tab is clicked, the action procedure makes the opposite occur.



The simplest way to create a tabbed user interface is to create an over-sized panel with all the controls visible and outside of 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 the controls into the tab control, get a list of the non-tab control names:

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

Determine which controls are to be visible in which tabs:

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(tca) : TabControl

   STRUCT WMTabControlAction &tca

   switch (tca.eventCode)
      case 2:                              // Mouse up
         Variable tabNum = tca.tab         // Active tab number
```

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

         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
         break
   endswitch

   return 0
End
```

A more elegant method, useful when you have many controls, is to systematically name the controls inside each tab using a prefix or suffix that is unique to that tab, such as tab0_thisCheck, tab0_thatCheck, tab1_multVar. Then use the ModifyControlList operation to show and hide the controls. See the **ModifyControlList** operation for an example.

Assign the action procedure to the tab control:

```
TabControl tb, proc=TabProc
```

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

Verify that the action procedure correctly shows and hides controls as you click the tabs. When this works correctly, move the controls into their final positions, inside the tab control.

During this process, the "temporary selection" shortcut comes in handy. While you are in operate mode, pressing Command-Option (*Macintosh*) or Ctrl+Alt (*Windows*) temporarily switchs to select mode, allowing you to select and drag controls.

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

```
Function CreatePanel()
   DoWindow/K TabPanel
   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
   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-876 for a complete description and examples.

Creating TitleBox Controls

The TitleBox operation creates or modifies a TitleBox control. The control's text can be static or can be tied to a global string variable. See the **TitleBox** operation on page V-897 for a complete description and examples.

Creating ValDisplay Controls

The **ValDisplay** operation (page V-916) creates or modifies a value display control.

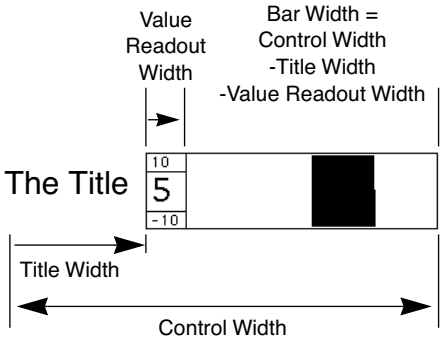
ValDisplay controls are very flexible and multifaceted. They can range from simple numeric readouts to thermometer bars or a hybrid of both. A ValDisplay control is tied to a numeric expression that you provide as an argument to the value keyword. Igor automatically updates the control whenever anything that the numeric expression depends on changes.

ValDisplay controls evaluate their value expression in the context of the root data folder. 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-916:

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

The size and 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* points or what is left. If the control width hasn't been used up, the bar is displayed in the remaining control width:



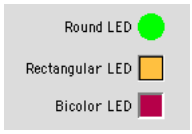
Here are the various major possible forms of ValDisplay controls. Some of these examples modify previous examples. For instance, the second bar-only example is a modification of the valdisp1 control created by the first bar-only example.

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

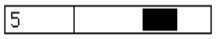
```
// Readout width = 0
ValDisplay valdisp1,frame=1,barmisc={12,0},limits={-10,10,0},value=K0
K0= 5                // halfway from base of 0 to high limit of 10.
```

The nice thing about a bar-only ValDisplay is that you can make it 5 to 200 points 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

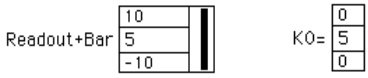
Whenever the numeric readout is visible, the optional limit values may be displayed too.

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

Optional Title

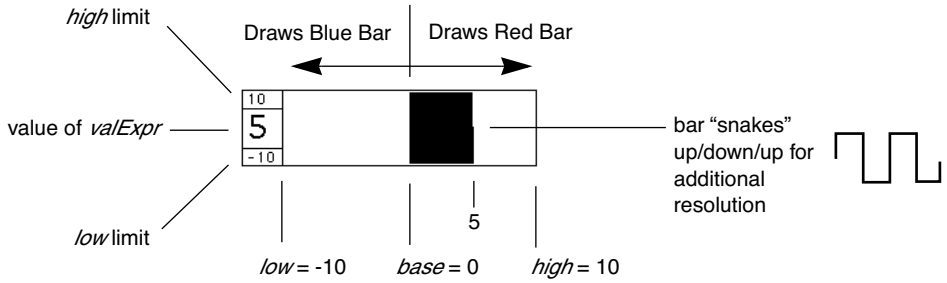
The control title steals horizontal space from the numeric readout and the bar, pushing them to the right. You may need to increase the control width to prevent them from disappearing.

```
// 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 Controls

You can kill (delete) a control from within a procedure using the **KillControl** operation (page V-408). 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.

Getting Information About Controls

You can use the **ControlInfo** operation (page V-74) 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.

ControlInfo is usually used for control panels that have a Do It button or equivalent. When the user clicks the button, its action procedure calls ControlInfo to query the state of each relevant control and acts accordingly.

ControlInfo is generally not used for the other style of control panel in which the action procedure for each control acts as soon as that control is clicked.

Updating Controls

You can use the **ControlUpdate** operation (page V-79) to cause a given control to redraw with its current value. You would use this in a procedure after changing the value or appearance of a control to display the changes before the normal update occurs.

Help Text for User-Defined Controls

Each control type has a help text property, set using the help keyword, through which you add a help tip. Tips are limited to 255 bytes.

Here is an example:

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

The tip appears when the user moves the mouse over the control, if tool tips are enabled in the Help section of the Miscellaneous Settings dialog.

Modifying Controls

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 a button control named `button0`, you can modify it with another `Button button0` command:

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

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

You can use the **ModifyControl** operation (page V-515) and **ModifyControlList** operation (page V-517) to modify a control without needing to know what kind of control it is:

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

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

Disabling and Hiding Controls

All controls support the keyword "disable=*d*" where *d* can be:

0:	Normal operation
1:	Hidden
2:	User input disabled
3:	Hidden and 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 is different from disable=2 mode in that noedit allows user input via the up or down arrows but disable=2 does not.

Control Background Color

The background color of control panel windows and the area at the top of a graph as reserved by the **ControlBar** operation (page V-73) is a shade of gray chosen to match the operating system look. This gray is used when the control bar background color, as set by ModifyGraph cbRGB or ModifyPanel cbRGB, is the default pure white, where the red, green and blue components are all 65535. Any other cbRGB setting, including not quite pure white, is honored. However, some controls or portions of controls are drawn by the operating system and may look out of place if you choose a different background color.

For special purposes, you can specify a background color for an individual control using the labelBack keyword. See the reference help of the individual control types for details.

Control Structures

Control action procedures can take one of two forms: structure-based or an old form that is not recommended. This section assumes that you are using the structure-based form.

The action procedure for a control uses a predefined, built-in structure as a parameter to the function. The procedure has this format:

```
Function ActionProcName(s)
   STRUCT <WMControlTypeActio>& s   // <WMControlTypeActio> is one of the
   …                                // structures listed below
End
```

The names of the various control structures are:

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 in the future. 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.

You can use the same action procedure for different controls of the same type, for all the buttons in one window, for example. Use the ctrlName field of the structure to identify the control and the win field to identify the window containing the control.

Control Structure Example

This example illustrates the extended event codes available for a button control. The function prints various text messages to the history area of the command window, depending what actions you take while in the button area.

```
Function ControlStructureTest()
   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	A mouse button is down.
Bit 1	Shift key is down.
Bit 2	Option (Macintosh) or Alt (Windows) is down.
Bit 3	Command (Macintosh) or Ctrl (Windows) is down.
Bit 4	Contextual menu click occurred.

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

Control Structure blockReentry Field

Because of architectural differences, reentry of the action procedure occurs on Macintosh only in Igor6 and does not occur at all in Igor7. Because reentry may be affected by internal changes in Igor or by operating system changes, it may reappear as an issue in the future.

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 only in Igor6 and does not occur at all in Igor7. Because reentry may be affected by internal changes in Igor or by operating system changes, it may reappear as an issue in the future.

```
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 string that is used by default. You can also store an unlimited number of additional user data strings by specifying a name for each one. The name can be any legal standard Igor name.

You can retrieve information from the default user data using the **ControlInfo** operation (page V-74), which returns such information in the S_UserData string variable. To retrieve any named user data, you must use the **GetUserData** operation (page V-273).

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 strings can slow down experiment saving and loading

User data is intended to replace or reduce the usage of global variables for maintaining state information related to controls.

Control User Data Examples

:Here is a simple example of a button with user data:

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

Here is a more complex example.

Copy the following code into the procedure window of a new experiment and run the `Panel0` macro. Then click 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
```

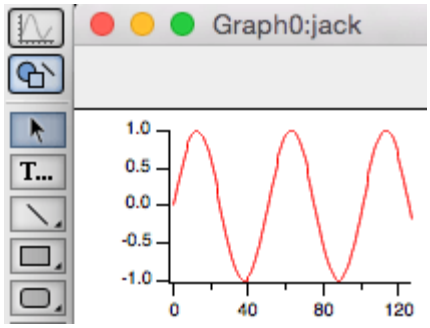
Action Procedures for Multiple Controls

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.

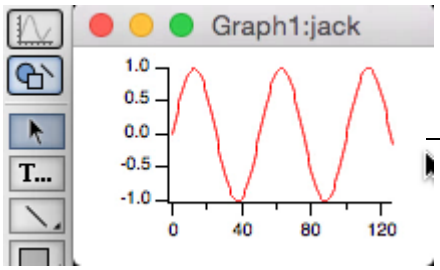
Controls in Graphs

The combination of controls and graphs provides a nice user interface for tinkering with data. You can create such a user interface by embedding controls in a graph or by embedding a graph in a control panel. This section explains the former technique, but the latter technique is usually recommended. See Chapter III-4, **Embedding and Subwindows** for details.

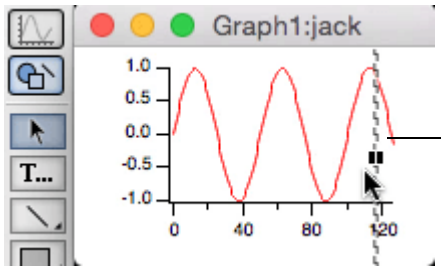
Although controls can be placed anywhere in a graph, you can and should reserve an area just for controls at the edge of a graph window. Controls in graphs operate much more smoothly if they reside in these reserved areas. The **ControlBar** operation (page V-73) or the Control Bar dialog can be used to set the height of a nonembedded control area at the top of the graph.



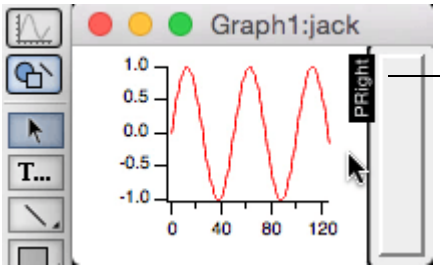
The simplest way to add a panel is to click near the edge of the graph and drag out a control area:



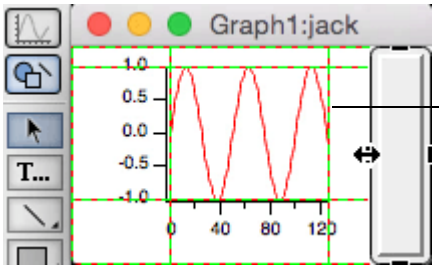
Click at the top, right, bottom, or left edge of the graph.



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 snaps and attaches 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, 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 **Control Background Color** on page III-388 for 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 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-79). 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 **Creating PopupMenu Controls** on page III-379.

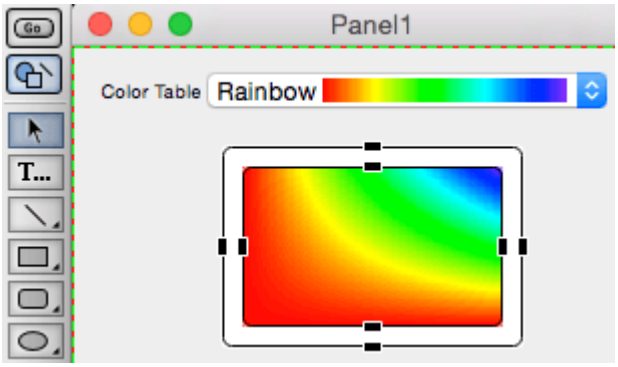
Control Panels

Control panels are windows designed to contain controls. The **NewPanel** creates a control panel.

Drawing tools can be used in panel windows to decorate control panels. Control panels have two drawing layers, UserBack and ProgBack, behind the controls and one layer, Overlay, in front of the controls. See **Drawing Layers** on page III-68 for details.

A panel window's 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 **Control Background Color** on page III-388 for 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 Subwindows** for details. This technique is cleaner than adding control areas to a graph. It also allows you to embed multiple graphs in one window with controls.

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-86 for details.

Exterior Control Panels

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.

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 Control 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.

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-18, **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 whether or not the drawing tools palette is initially shown or hidden when a new panel is created.

Controls Shortcuts

Action	Shortcut (*Macintosh*)	Shortcut (*Windows*)
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.