Chapter

IV-6

Interacting with the User

Overview

The following sections describe the various programming techniques available for getting input from and for interacting with a user during the execution of your procedures. These techniques include:

- The simple input dialog
- Control panels
- Cursors
- Marquee menus

The simple input dialog provides a bare bones but functional user interfaces with just a little programming. In situations where more elegance is required, control panels provide a better solution.

Modal and Modeless User Interface Techniques

Before the rise of the graphical user interface, computer programs worked something like this:

1. The program prompts the user for input.
2. The user enters the input.
3. The program does some processing.
4. Return to step 1.

In this model, the program is in charge and the user must respond with specific input at specific points of program execution. This is called a “modal” user interface because the program has one mode in which it will only accept specific input and another mode in which it will only do processing.

The Macintosh changed all this with the idea of event-driven programming. In this model, the computer waits for an event such as a mouse click or a key press and then acts on that event. The user is in charge and the program responds. This is called a “modeless” user interface because the program will accept any user action at any time.

You can use both techniques in Igor. Your program can put up a modal dialog asking for input and then do its processing or you can use control panels to build a sophisticated modeless event-driven system.

Event-driven programming is quite a bit more work than dialog-driven programming. You have to be able to handle user actions in any order rather than progressing through a predefined sequence of steps. In real life, a combination of these two methods is often used.

The Simple Input Dialog

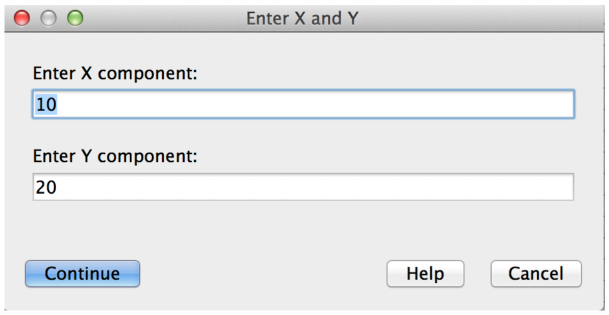
The simple input dialog is a way by which a function can get input from the user in a modal fashion. It is very simple to program and is also simple in appearance.

A simple input dialog is presented to the user when a DoPrompt statement is executed in a function. Parameters to DoPrompt specify the title for the dialog and a list of local variables. For each variable, you must include a Prompt statement that provides the text label for the variable.

Generally, the simple input dialog is used in conjunction with routines that run when the user chooses an item from a menu. This is illustrated in the following example which you can type into the procedure window of a new experiment:

```
Menu "Macros"
   "Calculate Diagonal...", CalcDiagDialog()
End

Function CalcDiagDialog()
   Variable x=10,y=20
   Prompt x, "Enter X component: "  // Set prompt for x param
   Prompt y, "Enter Y component: "  // Set prompt for y param
   DoPrompt "Enter X and Y", x, y
   if (V_Flag)
      return -1                     // User canceled
   endif

   Print "Diagonal=",sqrt(x^2+y^2)
End
```

If you run the CalcDiagDialog function, you see the following dialog:



Enter X component:
10

Enter Y component:
20

If the user presses Continue without changing the default values, "Diagonal= 22.3607" is printed in the history area of the command window. If the user presses Cancel, nothing is printed because DoPrompt sets the V_Flag variable to 1 in this case.

The simple input dialog allows for up to 10 numeric or string variables. When more than 5 items are used, the dialog uses two columns and you may have to limit the length of your Prompt text.

The simple input dialog is unique in that you can enter not only literal numbers or strings but also numeric expressions or string expressions. Any literal strings that you enter must be quoted.

If the user presses the Help button, Igor searches for a help topic with a name derived from the dialog title. If such a help topic is not found, then generic help about the simple input dialog is presented. In both cases, the input dialog remains until the user presses either Continue or Cancel.

Pop-Up Menus in Simple Dialogs

The simple input dialog supports pop-up menus as well as text items. The pop-up menus can contain an arbitrary list of items such as a list of wave names. To use a pop-up menu in place of the normal text entry item in the dialog, you use the following syntax in the prompt declaration:

```
Prompt <variable name>, <title string>, popup <menu item list>
```

The popup keyword indicates that you want a pop-up menu instead of the normal text entry item. The menu list specifies the items in the pop-up menu separated by semicolons. For example:

```
Prompt color, "Select Color", popup "red;green;blue;"
```

If the menu item list is too long to fit on one line, you can compose the list in a string variable like so:

```
String stmp= "red;green;blue;"
stmp += "yellow;purple"
Prompt color, "Select Color", popup stmp
```

The pop-up menu items support the same special characters as the user-defined menu definition (see **Special Characters in Menu Item Strings** on page IV-125) except that items in pop-up menus are limited to 50 characters, keyboard shortcuts are not supported, and special characters are disabled by default.

You can use pop-up menus with both numeric and string parameters. When used with numeric parameters the number of the item chosen is placed in the variable. Numbering starts from one. When used with string parameters the text of the chosen item is placed in the string variable.

There are a number of functions, such as the **WaveList** function (see page V-927) and the **TraceNameList** function (see page V-903), that are useful in creating pop-up menus.

To obtain a menu item list of all waves in the current data folder, use:

```
WaveList("*", ";", "")
```

To obtain a menu item list of all waves in the current data folder whose names end in “_X”, use:

```
WaveList("*_X", ";", "")
```

To obtain a menu item list of all traces in the top graph, use:

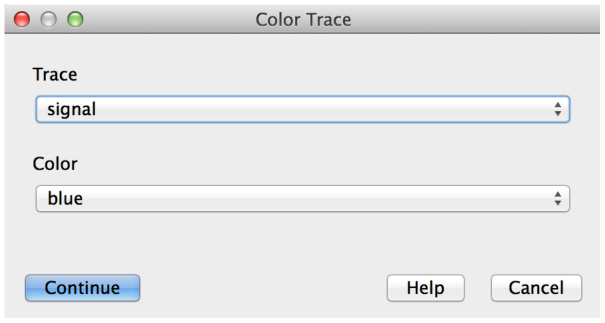
```
TraceNameList("", ";", 1)
```

For a list of all contours in the top graph, use ContourNameList. For a list of all images, use ImageNameList. For a list of waves in a table, use WaveList.

This next example creates two pop-up menus in the simple input dialog.

```
Menu "Macros"
   "Color Trace...", ColorTraceDialog()
End

Function ColorTraceDialog()
   String traceName
   Variable color=3
   Prompt traceName,"Trace",popup,TraceNameList("",";",1)
   Prompt color,"Color",popup,"red;green;blue"
   DoPrompt "Color Trace",traceName,color
   if( V_Flag )
      return 0          // user canceled
   endif

   if (color == 1)
      ModifyGraph rgb($traceName)=(65000, 0, 0)
   elseif(color == 2)
      ModifyGraph rgb($traceName)=(0, 65000, 0)
   elseif(color == 3)
      ModifyGraph rgb($traceName)=(0, 0, 65000)
   endif
End
```

If you choose Color Trace from the Macros menu, Igor displays the simple input dialog with two pop-up menus. The first menu contains a list of all traces in the target window which is assumed to be a graph. The second menu contains the items red, green and blue with blue (item number 3) initially chosen.



After you choose the desired trace and color from the pop-up menus and click the Continue button, the function continues execution. The string parameter traceName will contain the name of the trace chosen from the first pop-up menu. The numeric parameter color will have a value of 1, 2 or 3, corresponding to red, green and blue.

In the preceding example, we needed a trace name to pass to the ModifyGraph operation. In another common situation, we need a wave reference to operate on. For example:

```
Menu "Macros"
   "Smooth Wave In Graph...",SmoothWaveInGraphDialog()
End

Function SmoothWaveInGraphDialog()
   String traceName
   Prompt traceName,"Wave",popup,TraceNameList("",";",1)
   DoPrompt "Smooth Wave In Graph",traceName

   WAVE w = TraceNameToWaveRef("", traceName)
   Smooth 5, w
End
```

The traceName parameter alone is not sufficient to specify which wave we want to smooth because it does not identify in which data folder the wave resides. The TraceNameToWaveRef function returns a wave reference which solves this problem.

Saving Parameters for Reuse

It is possible to write a procedure that presents a simple input dialog with default values for the parameters saved from the last time it was invoked. To accomplish this, we use global variables to store the values between calls to the procedure. Here is an example that saves one numeric and one string variable.

```
Function TestDialog()
   String saveDF = GetDataFolder(1)
   NewDataFolder/O/S root:Packages
   NewDataFolder/O/S :TestDialog

   Variable num = NumVarOrDefault("gNum", 42)
   Prompt num, "Enter a number"
   String str = StrVarOrDefault("gStr", "Hello")
   Prompt str, "Enter a string"
   DoPrompt "test",num,str

   Variable/G gNum = num      // Save for next time
   String/G gStr = str

   // Put function body here
   Print num,str

   SetDataFolder saveDF
End
```

This example illustrates the NumVarOrDefault and StrVarOrDefault functions. These functions return the value of a global variable or a default value if the global variable does not exist. 42 is the default value for gNum. NumVarOrDefault returns 42 if gNum does not exist. If gNum does exist, it returns the value of gNum. Similarly, “Hello” is the default value for gStr. StrVarOrDefault returns “Hello” if gStr does not exist. If gStr does exist, it returns the value of gStr.

Multiple Simple Input Dialogs

Prompt statements can be located anywhere within the body of a function and they do not need to be grouped together, although it will aid code readability if associated Prompt and DoPrompt code is kept together. Functions may contain multiple DoPrompt statements, and Prompt statements can be reused or redefined.

The following example illustrates multiple simple input dialogs and prompt reuse:

```
Function Example()
   Variable a= 123
   Variable/C ca= cmplx(3,4)
   String s

   Prompt a,"Enter a value"
   Prompt ca,"Enter complex value"
   Prompt s,"Enter a string", popup "red;green;blue"
   DoPrompt "Enter Values",a,s,ca
   if(V_Flag)
      Abort "The user pressed Cancel"
   endif

   Print "a= ",a,"s= ",s,"ca=",ca

   Prompt a,"Enter a again please"
   Prompt s,"Type a string"
   DoPrompt "Enter Values Again", a,s

   if(V_Flag)
      Abort "The user pressed Cancel"
   endif

   Print "Now a=",a," and s=",s
End
```

When this function is executed, it produces two simple input dialogs, one after the other after the user clicks Continue.

Help For Simple Input Dialogs

You can create, for each simple input dialog, custom help that appears when the user clicks the Help button. You do so by providing a custom help file with topics that correspond to the titles of your dialogs as specified in the DoPrompt commands.

If there is no exactly matching help topic or subtopic for a given dialog title, Igor munges the presumptive topic by replacing underscore characters with spaces and inserting spaces before capital letters in the interior of the topic. For example, if the dialog title is “ReallyCoolFunction”, and there is no matching help topic or subtopic, Igor looks for a help topic or subtopic named “Really Cool Function”.

See **Creating Your Own Help File** on page IV-241 for information on creating custom help files.

Displaying an Open File Dialog

You can display an Open File dialog to allow the user to choose a file to be used with a subsequent command. For example, the user can choose a file which you will then use in a **LoadWave** command. The Open File dialog is displayed using an **Open**/D/R command. Here is an example:

```
Function/S DoOpenFileDialog()
   Variable refNum
   String message = "Select a file"
   String outputPath
   String fileFilters = "Data Files (*.txt,*.dat,*.csv):.txt,.dat,.csv;"
   fileFilters += "All Files:.*;"

   Open /D /R /F=fileFilters /M=message refNum
   outputPath = S_fileName

   return outputPath    // Will be empty if user canceled
End
```

Here the Open operation does not actually open a file but instead displays an Open File dialog. If the user chooses a file and clicks the Open button, the Open operation returns the full path to the file in the S_fileName output string variable. If the user cancels, Open sets S_fileName to "".

The /M flag is used to set the prompt message. As of OS X 10.11, Apple no longer shows the prompt message in the Open File dialog. It continues to work on Windows.

The /F flag is used to control the file filter which determines what kinds of files the user can select. This is explained further under **Open File Dialog File Filters**.

Prompt Does Not Work on Macintosh

As of OS X 10.11, Apple no longer shows the title bar in the Open File, Choose Folder, and Create Folder dialogs. Consequently, the prompt specified by the /M flag, which was displayed in the title bar, is no longer visible. Apple still shows the title bar in the Save File dialog.

Displaying a Multi-Selection Open File Dialog

You can display an Open File dialog to allow the user to choose multiple files to be used with subsequent commands. The multi-selection Open File dialog is displayed using an **Open**/D/R/MULT=1 command. The list of files selected is returned via S_fileName in the form of a carriage-return-delimited list of full paths.

Here is an example:

```
Function/S DoOpenMultiFileDialog()
   Variable refNum
   String message = "Select one or more files"
   String outputPaths
   String fileFilters = "Data Files (*.txt,*.dat,*.csv):.txt,.dat,.csv;"
   fileFilters += "All Files:.*;"

   Open /D /R /MULT=1 /F=fileFilters /M=message refNum
   outputPaths = S_fileName

   if (strlen(outputPaths) == 0)
      Print "Cancelled"
   else
      Variable numFilesSelected = ItemsInList(outputPaths, "\r")
      Variable i
      for(i=0; i<numFilesSelected; i+=1)
         String path = StringFromList(i, outputPaths, "\r")
         Printf "%d: %s\r", i, path
      endfor
   endif

   return outputPaths   // Will be empty if user canceled
End
```

Here the Open operation does not actually open a file but instead displays an Open File dialog. Because /MULT=1 was used, if the user chooses one or more files and clicks the Open button, the Open operation returns the list of full paths to files in the S_fileName output string variable. If the user cancels, Open sets S_fileName to "".

The list of full paths is delimited with a carriage return character, represented by "\r" in the example above. We use carriage return as the delimiter because the customary delimiter, semicolon, is a legal character in a Macintosh file name.

The /M flag is used to set the prompt message. As of OS X 10.11, Apple no longer shows the prompt message in the Open File dialog. It continues to work on Windows.

The /F flag is used to control the file filter which determines what kinds of files the user can select. This is explained further under **Open File Dialog File Filters**.

Open File Dialog File Filters

The **Open** operation displays the open file dialog if you use the /D/R flags or if the file to be opened is not fully specified using the pathName and fileNameStr parameters. The Open File dialog includes a file filter menu that allows the user to choose the type of file to be opened. By default this menus contain "Plain Text Files" and "All Files". You can use the /T and /F flags to override the default filter behavior.

The /T flag uses obsolescent Macintosh file types or file name extensions consisting of a dot plus three characters. The /F flag, added in Igor Pro 6.10, supports file name extensions only (not Macintosh file types) and extensions can be from one to 31 characters. Procedures written for Igor Pro 6.10 or later should use the /F flag in most cases but can use /T or both /T and /F. Procedures that must run with Igor Pro 6.0x and earlier must use the /T flag.

Using the /T=typeStr flag, you specify acceptable Macintosh-style file types represented by four-character codes (e.g., "TEXT") or acceptable three-character file name extensions (e.g., ".txt"). The pattern "????" means "any type of file" and is represented by "All Files" in the filter menu.

typeStr may contain multiple file types or extensions (e.g., "TEXTEPSF????" or ".txt.eps????"). Each file type or extension must be exactly four characters in length. Consequently the /T flag can accommodate only three-character file name extensions. Each file type or extension creates one entry in the Open File dialog filter menu.

If you use the /T flag, the Open operation automatically adds a filter for All Files ("????") if you do not add one explicitly.

Igor maps Macintosh file types to extensions. For example, if you specify /T="TEXT", you can open files with the extension ".txt" as well as any file whose Macintosh file type property is 'TEXT'. Igor does similar mappings for other extensions. See **File Types and Extensions** on page III-404 for details.

Using the /F=fileFilterStr flag, you specify a filter menu string plus acceptable file name extensions for each filter. fileFilterStr specifies one or more filters in a semicolon-separated list. For example, this specifies three filters:

```
String fileFilters = "Data Files (*.txt,*.dat,*.csv):.txt,.dat,.csv;"
fileFilters += "HTML Files (*.htm,*.html):.htm,.html;"
fileFilters += "All Files:.*;"
Open /F=fileFilters . . .
```

Each file filter consists of a filter menu string (e.g., "Data Files") followed by a colon, followed by one or more file name extensions (e.g., ".txt,.dat,.csv") followed by a semicolon. The syntax is rigid - no extra characters are allowed and the semicolons shown above are required. In this example the filter menu would contain "Data Files" and would accept any file with a ".txt", ".dat", or ".csv" extension. ".*" creates a filter that accepts any file.

If you use the /F flag, it is up to you to add a filter for All Files as shown above. It is recommended that you do this.

Displaying a Save File Dialog

You can display a Save File dialog to allow the user to choose a file to be created or overwritten by a subsequent command. For example, the user can choose a file which you will then create or overwrite via a Save command. The Save File dialog is displayed using an **Open**/D command. Here is an example:

```
Function/S DoSaveFileDialog()
   Variable refNum
   String message = "Save a file"
   String outputPath
   String fileFilters = "Data Files (*.txt):.txt;"
   fileFilters += "All Files:.*;"

   Open /D /F=fileFilters /M=message refNum
   outputPath = S_fileName

   return outputPath    // Will be empty if user canceled
End
```

Here the Open operation does not actually open a file but instead displays a Save File dialog. If the user chooses a file and clicks the Save button, the Open operation returns the full path to the file in the S_fileName output string variable. If the user cancels, Open sets S_fileName to "".

The /M flag is used to set the prompt message. As of OS X 10.11, Apple no longer shows the prompt message in the Save File dialog. It continues to work on Windows.

The /F flag is used to control the file filter which determines what kinds of files the user can create. This is explained further under **Save File Dialog File Filters**.

Save File Dialog File Filters

The Save File dialog includes a file filter menu that allows the user to choose the type of file to be saved. By default this menus contain "Plain Text File" and, on Windows only, "All Files". You can use the /T and /F flags to override the default filter behavior.

The /T and /F flags work as explained under Open File Dialog File Filters. Using the /F flag for a Save File dialog, you would typically specify just one filter plus All Files, like this:

String fileFilters = "Data File (*.dat):.dat;"

fileFilters += "All Files:.*;"

Open /F=fileFilters . . .

The file filter chosen in the Save File dialog determines the extension for the file being saved. For example, if the "Plain Text Files" filter is selected, the ".txt" extension is added if you don't explicitly enter it in the File Name edit box. However if you select the "All Files" filter then no extension is automatically added and the final file name is whatever you enter in the File Name edit box. You should include the "All Files" filter if you want the user to be able to specify a file name with any extension. If you want to force the file name extension to an extension of your choice rather than the user's, omit the "All Files" filter.

Using Open in a Utility Routine

To be as general and useful as possible, a utility routine that acts on a file should have a pathName parameter and a fileName parameter, like this:

```
Function ShowFileInfo(pathName, fileName)
   String pathName   // Name of symbolic path or "" for dialog.
   String fileName   // File name or "" for dialog.

   <Show file info here>
End
```

This provides flexibility to the calling function. The caller can supply a valid symbolic path name and a simple leaf name in fileName, a valid symbolic path name and a partial path in fileName, or a full path in fileName in which case pathName is irrelevant.

If pathName and fileName fully specify the file of interest, you want to just open the file and perform the requested action. However, if pathName and fileName do not fully specify the file of interest, you want to display an Open File dialog so the user can choose the file. This is accomplished by using the **Open** operation's /D=2 flag.

With /D=2, if pathName and fileName fully specify the file, the Open operation merely sets the S_fileName output string variable to the full path to the file. If pathName and fileName do not fully specify the file, Open displays an Open File dialog and then sets the S_fileName output string variable to the full path to the file. If the user cancels the Open File dialog, Open sets S_fileName to "". In all cases, Open/D=2 just sets S_fileName and does not actually open the file.

If pathName and fileName specify an alias (*Macintosh*) or shortcut (*Windows*), Open/D=2 returns the file referenced by the alias or shortcut.

Here is how you would use Open /D=2.

```
Function ShowFileInfo(pathName, fileName)
   String pathName      // Name of symbolic path or "" for dialog.
   String fileName      // File name or "" for dialog.

   Variable refNum

   Open /D=2 /R /P=$pathName refNum as fileName    // Sets S_fileName

   if (strlen(S_fileName) == 0)
      Print "ShowFileInfo was canceled"
   else
      String fullPath = S_fileName
      Print fullPath
      Open /R refNum as fullPath
      FStatus refNum    // Sets S_info
      Print S_info
      Close refNum
   endif
End
```

In this case, we wanted to open the file for reading. To create a file and open it for writing, omit /R from both calls to Open.

Pause For User

The **PauseForUser** operation (see page V-626) allows an advanced programmer to create a more sophisticated semimodal user interface. When you invoke it from a procedure, Igor suspends procedure execution and the user can interact with graph, table or control panel windows using the mouse or keyboard. Execution continues when the user kills the main window specified in the PauseForUser command.

PauseForUser is fragile because it attempts to create a semi-modal mode which is not supported by the operating system. For code that needs to be bulletproof, use an alternative modeless approach, if possible.

Pausing execution can serve two purposes. First, the programmer can pause function execution so that the user can, for example, adjust cursors in a graph window before continuing with a curve fit. In this application, the programmer creates a control panel with a continue button that the user presses after adjusting the cursors in the target graph. Pressing the continue button kills the host control panel (see example below).

In the second application, the programmer may wish to obtain input from the user in a more sophisticated manner than can be done using DoPrompt commands. This method uses a control panel as the main window with no optional target window. It is similar to the control panel technique shown above, except that it is modal.

Following are some examples of how you can use the **PauseForUser** operation (see page V-626) in your own user functions.

PauseForUser Simple Cursor Example

This example shows how to allow the user to adjust cursors on a graph while a procedure is executing. Most of the work is done by the UserCursorAdjust function. UserCursorAdjust is called by the Demo function which first creates a graph and shows the cursor info panel.

This example illustrates two modes of PauseForUser. When called with autoAbortSecs=0, UserCursorAdjust calls PauseForUser without the /C flag in which case PauseForUser retains control until the user clicks the Continue button.

When called with autoAbortSecs>0, UserCursorAdjust calls PauseForUser/C. This causes PauseForUser to handle any pending events and then return to the calling procedure. The procedure checks the V_flag variable, set by PauseForUser, to determine when the user has finished interacting with the graph. PauseForUser/C, which requires Igor Pro 6.1 or later, is for situations where you want to do something while the user interacts with the graph.

To try this yourself, copy and paste all three routines below into the procedure window of a new experiment and then run the Demo function with a value of 0 and again with a value such as 30.

```
Function UserCursorAdjust(graphName,autoAbortSecs)
   String graphName
   Variable autoAbortSecs

   DoWindow/F $graphName                    // Bring graph to front
   if (V_Flag == 0)                         // Verify that graph exists
      Abort "UserCursorAdjust: No such graph."
      return -1
   endif

   NewPanel /K=2 /W=(187,368,437,531) as "Pause for Cursor"
   DoWindow/C tmp_PauseforCursor            // Set to an unlikely name
   AutoPositionWindow/E/M=1/R=$graphName  // Put panel near the graph

   DrawText 21,20,"Adjust the cursors and then"
   DrawText 21,40,"Click Continue."
   Button button0,pos={80,58},size={92,20},title="Continue"
   Button button0,proc=UserCursorAdjust_ContButtonProc
   Variable didAbort= 0
   if( autoAbortSecs == 0 )
      PauseForUser tmp_PauseforCursor,$graphName
   else
      SetDrawEnv textyjust= 1
      DrawText 162,103,"sec"
      SetVariable sv0,pos={48,97},size={107,15},title="Aborting in "
      SetVariable sv0,limits={-inf,inf,0},value= _NUM:10
      Variable td= 10,newTd
      Variable t0= ticks
      Do
         newTd= autoAbortSecs - round((ticks-t0)/60)
         if( td != newTd )
            td= newTd
            SetVariable sv0,value= _NUM:newTd,win=tmp_PauseforCursor
            if( td <= 10 )
               SetVariable sv0,valueColor= (65535,0,0),win=tmp_PauseforCursor
            endif
         endif
         if( td <= 0 )
            DoWindow/K tmp_PauseforCursor
            didAbort= 1
            break
         endif

         PauseForUser/C tmp_PauseforCursor,$graphName
      while(V_flag)
   endif
   return didAbort
End

Function UserCursorAdjust_ContButtonProc(ctrlName) : ButtonControl
   String ctrlName

   DoWindow/K tmp_PauseforCursor              // Kill panel
End

Function Demo(autoAbortSecs)
```

```
   Variable autoAbortSecs

   Make/O jack;SetScale x,-5,5,jack
   jack= exp(-x^2)+gnoise(0.1)
   DoWindow Graph0
   if( V_Flag==0 )
      Display jack
      ShowInfo
   endif

   if (UserCursorAdjust("Graph0",autoAbortSecs) != 0)
      return -1
   endif

   if (strlen(CsrWave(A))>0 && strlen(CsrWave(B))>0)// Cursors are on trace?
      CurveFit gauss,jack[pcsr(A),pcsr(B)] /D
   endif
End
```

PauseForUser Advanced Cursor Example

Now for something a bit more complex. Here we modify the preceding example to include a Cancel button. For this, we need to return information about which button was pressed. Although we could do this by creating a single global variable in the root data folder, we use a slightly more complex technique using a temporary data folder. This technique is especially useful for more complex panels with multiple output variables because it eliminates name conflict issues. It also allows much easier clean up because we can kill the entire data folder and everything in it with just one operation.

```
Function UserCursorAdjust(graphName)
   String graphName

   DoWindow/F $graphName   // Bring graph to front
   if (V_Flag == 0)        // Verify that graph exists
      Abort "UserCursorAdjust: No such graph."
      return -1
   endif

   NewDataFolder/O root:tmp_PauseforCursorDF
   Variable/G root:tmp_PauseforCursorDF:canceled= 0

   NewPanel/K=2 /W=(139,341,382,450) as "Pause for Cursor"
   DoWindow/C tmp_PauseforCursor          // Set to an unlikely name
   AutoPositionWindow/E/M=1/R=$graphName  // Put panel near the graph

   DrawText 21,20,"Adjust the cursors and then"
   DrawText 21,40,"Click Continue."
   Button button0,pos={80,58},size={92,20},title="Continue"
   Button button0,proc=UserCursorAdjust_ContButtonProc
   Button button1,pos={80,80},size={92,20}
   Button button1,proc=UserCursorAdjust_CancelBProc,title="Cancel"

   PauseForUser tmp_PauseforCursor,$graphName

   NVAR gCaneled= root:tmp_PauseforCursorDF:canceled
   Variable canceled= gCaneled   // Copy from global to local
                                 // before global is killed
   KillDataFolder root:tmp_PauseforCursorDF

   return canceled
End

Function UserCursorAdjust_ContButtonProc(ctrlName) : ButtonControl
   String ctrlName
   DoWindow/K tmp_PauseforCursor    // Kill self
End

Function UserCursorAdjust_CancelBProc(ctrlName) : ButtonControl
   String ctrlName

   Variable/G root:tmp_PauseforCursorDF:canceled= 1
   DoWindow/K tmp_PauseforCursor    // Kill self
End

Function Demo()
   Make/O jack;SetScale x,-5,5,jack
   jack= exp(-x^2)+gnoise(0.1)
   DoWindow Graph0
   if (V_Flag==0)
      Display jack
      ShowInfo
   endif
   Variable rval= UserCursorAdjust("Graph0")
   if (rval == -1)            // Graph name error?
      return -1;
   endif
   if (rval == 1)             // User canceled?
      DoAlert 0,"Canceled"
      return -1;
   endif
   CurveFit gauss,jack[pcsr(A),pcsr(B)] /D
End
```

PauseForUser Control Panel Example

This example illustrates using a control panel as a modal dialog via PauseForUser. This technique is useful when you need a more sophisticated modal user interface than provided by the simple input dialog.

We started by manually creating a control panel. When the panel design was finished, we closed it to create a recreation macro. We then used code copied from the recreation macro in the DoMyInputPanel function and deleted the recreation macro.

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

   DoWindow/K tmp_GetInputPanel     // kill self
End

// Call with these variables already created and initialized:
//    root:tmp_PauseForUserDemo:numvar
//    root:tmp_PauseForUserDemo:strvar
Function DoMyInputPanel()
   NewPanel /W=(150,50,358,239)
   DoWindow/C tmp_GetInputPanel        // set to an unlikely name
   DrawText 33,23,"Enter some data"
   SetVariable setvar0,pos={27,49},size={126,17},limits={-Inf,Inf,1}
   SetVariable setvar0,value= root:tmp_PauseForUserDemo:numvar
   SetVariable setvar1,pos={24,77},size={131,17},limits={-Inf,Inf,1}
   SetVariable setvar1,value= root:tmp_PauseForUserDemo:strvar
   Button button0,pos={52,120},size={92,20}
   Button button0,proc=UserGetInputPanel_ContButton,title="Continue"

   PauseForUser tmp_GetInputPanel
End

Function Demo1()
   NewDataFolder/O root:tmp_PauseForUserDemo
   Variable/G root:tmp_PauseForUserDemo:numvar= 12
   String/G root:tmp_PauseForUserDemo:strvar= "hello"
   DoMyInputPanel()

   NVAR numvar= root:tmp_PauseForUserDemo:numvar
   SVAR strvar= root:tmp_PauseForUserDemo:strvar

   printf "You entered %g and %s\r",numvar,strvar

   KillDataFolder root:tmp_PauseForUserDemo
End
```

Progress Windows

Sometimes when performing a long calculation, you may want to display an indication that the calculation is in progress, perhaps showing how far along it is, and perhaps providing an abort button. As of Igor Pro 6.1, you can use a control panel window for this task using the **DoUpdate** /E and /W flags and the mode=4 setting for **ValDisplay**.

DoUpdate /W=win /E=1 marks the specified window as a progress window that can accept mouse events while user code is executing. The /E flag need be used only once to mark the panel but it does not hurt to use it in every call. This special state of the control panel is automatically cleared when procedure execution finishes and Igor's outer loop again runs.

For a window marked as a progress window, DoUpdate sets V_Flag to 2 if a mouse up happened in a button since the last call. When this occurs, the full path to the subwindow containing the button is stored in S_path and the name of the control is stored in S_name.

Here is a simple example that puts up a progress window with a progress bar and a Stop button. Try each of the four input flag combinations.

```
//    ProgressDemo1(0,0)
//    ProgressDemo1(1,0)
//    ProgressDemo1(0,1)
//    ProgressDemo1(1,1)
Function ProgressDemo1(indefinite, useIgorDraw)
   Variable indefinite
   Variable useIgorDraw// True to use Igor's own draw method rather than native

   NewPanel /N=ProgressPanel /W=(285,111,739,193)
   ValDisplay valdisp0,pos={18,32},size={342,18}
   ValDisplay valdisp0,limits={0,100,0},barmisc={0,0}
   ValDisplay valdisp0,value= _NUM:0
   if( indefinite )
      ValDisplay valdisp0,mode= 4// candy stripe
   else
      ValDisplay valdisp0,mode= 3// bar with no fractional part
   endif
   if( useIgorDraw )
      ValDisplay valdisp0,highColor=(0,65535,0)
   endif
   Button bStop,pos={375,32},size={50,20},title="Stop"
   DoUpdate /W=ProgressPanel /E=1// mark this as our progress window

   Variable i,imax= indefinite ? 10000 : 100
   for(i=0;i<imax;i+=1)
      Variable t0= ticks
      do
      while( ticks < (t0+3) )
      if( indefinite )
         ValDisplay valdisp0,value= _NUM:1,win=ProgressPanel
      else
         ValDisplay valdisp0,value= _NUM:i+1,win=ProgressPanel
      endif
      DoUpdate /W=ProgressPanel
      if( V_Flag == 2 )// we only have one button and that means stop
         break
      endif
   endfor
   KillWindow ProgressPanel
End
```

When performing complex calculations, it is often difficult to insert DoUpdate calls in the code. In this case, you can use a window hook that responds to event #23, spinUpdate. This is called at the same time that the beachball icon in the status bar spins. The hook can then update the window's control state and then call DoUpdate/W on the window. If the window hook returns non-zero, then an abort is performed. If you desire a more controlled quit, you might set a global variable that your calculation code can test

The following example provides an indefinite indicator and an abort button. Note that if the abort button is pressed, the window hook kills the progress window since otherwise the abort would cause the window to remain.

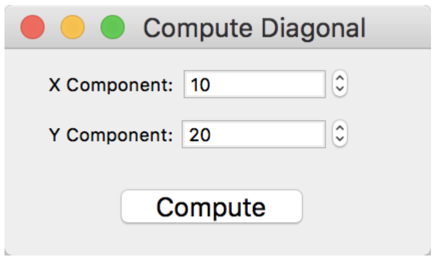
```
// Example: ProgressDemo2(100)
Function ProgressDemo2(nloops)
   Variable nloops

   Variable useIgorDraw=0  // set true for Igor draw method rather than native

   NewPanel/FLT /N=myProgress/W=(285,111,739,193) as "Calculating..."
   ValDisplay valdisp0,pos={18,32},size={342,18}
   ValDisplay valdisp0,limits={0,100,0},barmisc={0,0}
   ValDisplay valdisp0,value= _NUM:0
   ValDisplay valdisp0,mode=4    // candy stripe
   if( useIgorDraw )
      ValDisplay valdisp0,highColor=(0,65535,0)
   endif
   Button bStop,pos={375,32},size={50,20},title="Abort"
   SetActiveSubwindow _endfloat_
   DoUpdate/W=myProgress/E=1     // mark this as our progress window

   SetWindow myProgress,hook(spinner)=MySpinnHook

   Variable t0= ticks,i
   for(i=0;i<nloops;i+=1)
      PerformLongCalc(1e6)
   endfor
   Variable timeperloop= (ticks-t0)/(60*nloops)

   KillWindow myProgress

   print "time per loop=",timeperloop
End

Function MySpinnHook(s)
   STRUCT WMWinHookStruct &s

   if( s.eventCode == 23 )
      ValDisplay valdisp0,value= _NUM:1,win=$s.winName
      DoUpdate/W=$s.winName
      if( V_Flag == 2 )    // we only have one button and that means abort
         KillWindow $s.winName
         return 1
      endif
   endif
   return 0
End

Function PerformLongCalc(nmax)
   Variable nmax

   Variable i,s
   for(i=0;i<nmax;i+=1)
      s+= sin(i/nmax)
   endfor
End
```

Control Panels and Event-Driven Programming

The CalcDiagDialog function shown under **The Simple Input Dialog** on page IV-132 creates a modal dialog. “Modal” means that the function retains complete control until the user clicks Cancel or Continue. The user can not activate another window or choose a menu item until the dialog is dismissed.

This section shows how to implement the same functionality using a control panel as a modeless dialog. “Modeless” means that the user can activate another window or choose a menu item at any time. The modeless window accepts input whenever the user wants to enter it but does not block the user from accessing other windows.

The control panel looks like this:



The code implementing this control panel is given below. Before we look at the code, here is some explanation of the thinking behind it.

The X Component and Y Component controls are SetVariable controls. We attach each SetVariable control to a global variable so that, if we kill and later recreate the panel, the SetVariable control is restored to its previous state. In other words, we use global variables to remember settings across invocations of the panel. To keep the global variables from cluttering the user’s space, we bury them in a data folder located at root:Packages:DiagonalControlPanel.

The DisplayDiagonalControlPanel routine creates the data folder and the global variables if they do not already exist. DisplayDiagonalControlPanel creates the control panel or, if it already exists, just brings it to the front.

We added a menu item to the Macros menu so the user can easily invoke DisplayDiagonalControlPanel.

We built the control panel manually using techniques explained in Chapter III-14, **Controls and Control Panels**. Then we closed it so Igor would create a display recreation macro which we named DiagonalControlPanel. We then manually tweaked the macro to attach the SetVariable controls to the desired globals and to set the panel’s behavior when the user clicks the close button by adding the /K=1 flag.

Here are the procedures.

```
// Add a menu item to display the control panel.
Menu "Macros"
   "Display Diagonal Control Panel", DisplayDiagonalControlPanel()
End
```

```
// This is the display recreation macro, created by Igor
// and then manually tweaked. The parts that were tweaked
// are shown in bold. NOTE: Some lines are wrapped to fit on the page.
Window DiagonalControlPanel() : Panel
   PauseUpdate; Silent 1   // building window...

   NewPanel/W=(162,95,375,198)/K=1 as "Compute Diagonal"

   SetVariable XSetVar,pos={22,11},size={150,15},title="X Component:"
   SetVariable XSetVar,limits={-Inf,Inf,1},value=
                  root:Packages:DiagonalControlPanel:gXComponent

   SetVariable YSetVar,pos={22,36},size={150,15},title="Y Component:"
   SetVariable YSetVar,limits={-Inf,Inf,1},value=
                  root:Packages:DiagonalControlPanel:gYComponent

   Button ComputeButton,pos={59,69},size={90,20},
                       proc=ComputeDiagonalProc,title="Compute"
EndMacro

// This is the action procedure for the Compute button.
// We created it using the Button dialog.
Function ComputeDiagonalProc(ctrlName) : ButtonControl
   String ctrlName

   DFREF dfr = root:Packages:DiagonalControlPanel

   NVAR gXComponent = dfr:gXComponent
   NVAR gYComponent = dfr:gYComponent
   Variable diagonal
   diagonal = sqrt(gXComponent^2 + gYComponent^2)
   Printf "Diagonal=%g\r", diagonal
End

// This is the top level routine which makes sure that the globals
// and their enclosing data folders exist and then makes sure that
// the control panel is displayed.
Function DisplayDiagonalControlPanel()
   // If the panel is already created, just bring it to the front.
   DoWindow/F DiagonalControlPanel
   if (V_Flag != 0)
      return 0
   endif

   String dfSave = GetDataFolder(1)

   // Create a data folder in Packages to store globals.
   NewDataFolder/O/S root:Packages
   NewDataFolder/O/S root:Packages:DiagonalControlPanel

   // Create global variables used by the control panel.
   Variable xComponent = NumVarOrDefault(":gXComponent", 10)
   Variable/G gXComponent = xComponent
   Variable yComponent = NumVarOrDefault(":gYComponent", 20)
   Variable/G gYComponent = yComponent

   // Create the control panel.
   Execute "DiagonalControlPanel()"

   SetDataFolder dfSave
End
```

To try this example, copy all of the procedures and paste them into the procedure window of a new experiment. Close the procedure window to compile it and then choose Display Diagonal Control Panel from the Macros menu. Next enter values in the text entry items and click the Compute button. Close the control panel and then reopen it using the Display Diagonal Control Panel menu item. Notice that the values that you entered were remembered. Use the Data Browser to inspect the root:Packages:DiagonalControlPanel data folder.

Although this example is very simple, it illustrates the process of creating a control panel that functions as a modeless dialog. There are many more examples of this in the Examples folder. You can access them via the File→Example Experiments submenu.

See Chapter III-14, **Controls and Control Panels**, for more information on building control panels.

Detecting a User Abort

If you have written a user-defined function that takes a long time to execute, you may want to provide a way for the user to abort it. One solution is to display a progress window as discussed under **Progress Windows** on page IV-144.

Here is a simple alternative using the escape key:

```
Function PressEscapeToAbort(phase, title, message)
   Variable phase    // 0: Display control panel with message.
                     // 1: Test if Escape key is pressed.
                     // 2: Close control panel.
   String title      // Title for control panel.
   String message    // Tells user what you are doing.

   if (phase == 0)   // Create panel
      DoWindow/F PressEscapePanel
      if (V_flag == 0)
         NewPanel/K=1 /W=(100,100,350,200)
         DoWindow/C PressEscapePanel
         DoWindow/T PressEscapePanel, title
      endif
      TitleBox Message,pos={7,8},size={69,20},title=message
      String abortStr = "Press escape to abort"
      TitleBox Press,pos={6,59},size={106,20},title=abortStr
      DoUpdate
   endif

   if (phase == 1)   // Test for Escape key
      Variable doAbort = 0
      if (GetKeyState(0) & 32)      // Is Escape key pressed now?
         doAbort = 1
      else
         if (strlen(message) != 0)  // Want to change message?
            TitleBox Message,title=message
            DoUpdate
         endif
      endif
      return doAbort
   endif

   if (phase == 2)   // Kill panel
      DoWindow/K PressEscapePanel
   endif

   return 0
End

Function Demo()
   // Create panel
   PressEscapeToAbort(0, "Demonstration", "This is a demo")
   Variable startTicks = ticks
   Variable endTicks = startTicks + 10*60
   Variable lastMessageUpdate = startTicks

   do
      String message
      message = ""
      if (ticks>=lastMessageUpdate+60) // Time to update message?
         Variable remaining = (endTicks - ticks) / 60
         sprintf message, "Time remaining: %.1f seconds", remaining
         lastMessageUpdate = ticks
      endif

      if (PressEscapeToAbort(1, "", message))
         Print "Test aborted by Escape key."
         break
      endif
   while(ticks < endTicks)

   PressEscapeToAbort(2, "", "")       // Kill panel.
End
```

Creating a Contextual Menu

You can use the **PopupContextualMenu** operation to create a pop-up menu in response to a contextual click (control-click (*Macintosh*) or right-click). You would do this from a window hook function or from the action procedure for a control in a control panel.

In this example, we create a control panel with a list. When the user right-clicks on the list, Igor sends a mouse-down event to the listbox procedure, TickerListProc in this case. The listbox procedure uses the eventMod field of the WMListboxAction structure to determine if the click is a right-click. If so, it calls HandleTickerListRightClick which calls PopupContextualMenu to display the contextual menu.

```
Menu "Macros"

   "Show Demo Panel", ShowDemoPanel()
End

static Function HandleTickerListRightClick()
   String popupItems = ""
   popupItems += "Refresh;"

   PopupContextualMenu popupItems
   strswitch (S_selection)
      case "Refresh":
         DoAlert 0, "Here is where you would refresh the ticker list."
         break
   endswitch
End

Function TickerListProc(lba) : ListBoxControl
   STRUCT WMListboxAction &lba

   switch (lba.eventCode)
      case 1:                    // Mouse down
         if (lba.eventMod & 0x10)// Right-click?
            HandleTickerListRightClick()
         endif
         break
   endswitch
   return 0
End

Function ShowDemoPanel()
   DoWindow/F DemoPanel
   if (V_flag != 0)
      return 0 // Panel already exists.
   endif

   // Create panel data.
   Make/O/T ticketListWave = {{"AAPL","IBM","MSFT"}, {"90.25","86.40","17.17"}}

   // Create panel.
   NewPanel /N=DemoPanel /W=(321,121,621,321) /K=1
   ListBox TickerList,pos={48,16},size={200,100},fSize=12
   ListBox TickerList,listWave=root:ticketListWave
   ListBox TickerList,mode= 1,selRow= 0, proc=TickerListProc
End
```

Cursors as Input Device

You can use the cursors on a trace in a graph to identify the data to be processed.

The examples shown above using PauseForUser are modal - the user adjusts the cursors in the middle of procedure execution and can do nothing else. This technique is non-modal — the user is expected to adjust the cursors before invoking the procedure.

This function does a straight-line curve fit through the data between cursor A (the round cursor) and cursor B (the square cursor). This example is written to handle both waveform and XY data.

```
Function FitLineBetweenCursors()
   Variable isXY

   // Make sure both cursors are on the same wave.
   WAVE wA = CsrWaveRef(A)
   WAVE wB = CsrWaveRef(B)
   String dfA = GetWavesDataFolder(wA, 2)
   String dfB = GetWavesDataFolder(wB, 2)
   if (CmpStr(dfA, dfB) != 0)
      Abort "Both cursors must be on the same wave."
      return -1
   endif

   // Find the wave that the cursors are on.
   WAVE yWave = CsrWaveRef(A)

   // Decide if this is an XY pair.
   WAVE xWave = CsrXWaveRef(A)
   isXY = WaveExists(xWave)

   if (isXY)
      CurveFit line yWave(xcsr(A),xcsr(B)) /X=xWave /D
   else
      CurveFit line yWave(xcsr(A),xcsr(B)) /D
   endif
End
```

This technique is demonstrated in the **Fit Line Between Cursors** example experiment in the “Examples:Curve Fitting” folder.

Advanced programmers can set things up so that a hook function is called whenever the user adjusts the position of a cursor. For details, see **Cursors — Moving Cursor Calls Function** on page IV-316.

Marquee Menu as Input Device

A marquee is the dashed-line rectangle that you get when you click and drag diagonally in a graph or page layout. It is used for expanding and shrinking the range of axes, for selecting a rectangular section of an image, and for specifying an area of a layout. You can use the marquee as an input device for your procedures. This is a relatively advanced technique.

This menu definition adds a user-defined item to the graph marquee menu:

```
Menu "GraphMarquee"
   "Print Marquee Coordinates", PrintMarqueeCoords()
End
```

To add an item to the layout marquee menu, use LayoutMarquee instead of GraphMarquee.

When the user chooses Print Marquee Coordinates, the following function runs. It prints the coordinates of the marquee in the history area. It assumes that the graph has left and bottom axes.

```
Function PrintMarqueeCoords()
   String format
   GetMarquee/K left, bottom
   format = "flag: %g; left: %g; top: %g; right: %g; bottom: %g\r"
   printf format, V_flag, V_left, V_top, V_right, V_bottom
End
```

The use of the marquee menu as in input device is demonstrated in the **Marquee Demo** and **Delete Points from Wave** example experiments.

Polygon as Input Device

This technique is similar to the marquee technique except that you can identify a nonrectangular area. It is implemented using **FindPointsInPoly** operation (see page V-213).