User Guide to LabVIEW & APT

Contents

Chapter 1 Introduction to LabVIEW and APT

LabVIEW

LabVIEW is a graphical programming language that uses icons instead of lines of text to create applications. In contrast to text-based programming languages, where instructions determine program execution, LabVIEW uses dataflow programming, where the flow of data determines execution.

In LabVIEW, you build a user interface with a set of tools and objects.

The user interface is known as the front panel. You then add code using graphical representations of functions to control the front panel objects.

The block diagram contains this code. In some ways, the block diagram resembles a flowchart.

Refer to the National Instruments web site at www.ni.com for more general information about LabVIEW.

APT

The apt™ Suite of controllers includes a range of compact drivers, high power bench top controllers and 19" rack based units that, together control our range of precision stages and actuators, support motion control from the 10’s of cm to the nm regime. The product offering comprises stepper motor and DC motor controllers, closed loop and open loop piezo controllers, strain gauge readers, and solenoid drivers, together with a sophisticated feedback controller (NanoTrak™) that fully optimizes coupled optical powers in a wide range of alignment scenarios. All of our controllers are supported by unified PC based user and programming utilities (the apt™ software suite) that enables higher level custom applications to be constructed extremely effectively and quickly. Thanks to the USB connectivity implemented on all of our controller units, it is extremely easy to link multiple units together to realize a multiaxis motion control solution for many positioning and alignment needs.

APT within LabVIEW

Programmers of LabVIEW through the use of ActiveX technology, used within the APT platform, can communicate and control any APT controller. By simply specifying which controller to be used within code function calls, properties and events can be used to expose a controllers functionality.

Chapter 2 LabVIEW Programs

LabVIEW programs are called virtual instruments, or VIs, because their appearance and operation imitates physical instruments. Each VI uses functions that manipulate input from the user interface or other sources and display that information or move it to other files or other computers.

A VI contains the following three components:

- **Front panel** - Serves as the user interface.
- **Block diagram** - Contains the graphical source code that defines the functionality of the VI.
- **Icon and connector pane** - Identifies the VI so that you can use the VI inside another VI.

Note: A VI within another VI is called a subVI. A subVI corresponds to a subroutine in text-based programming languages.

Front Panel

The front panel is the user interface of the VI. Figure 2-1 shows an example of a front panel.

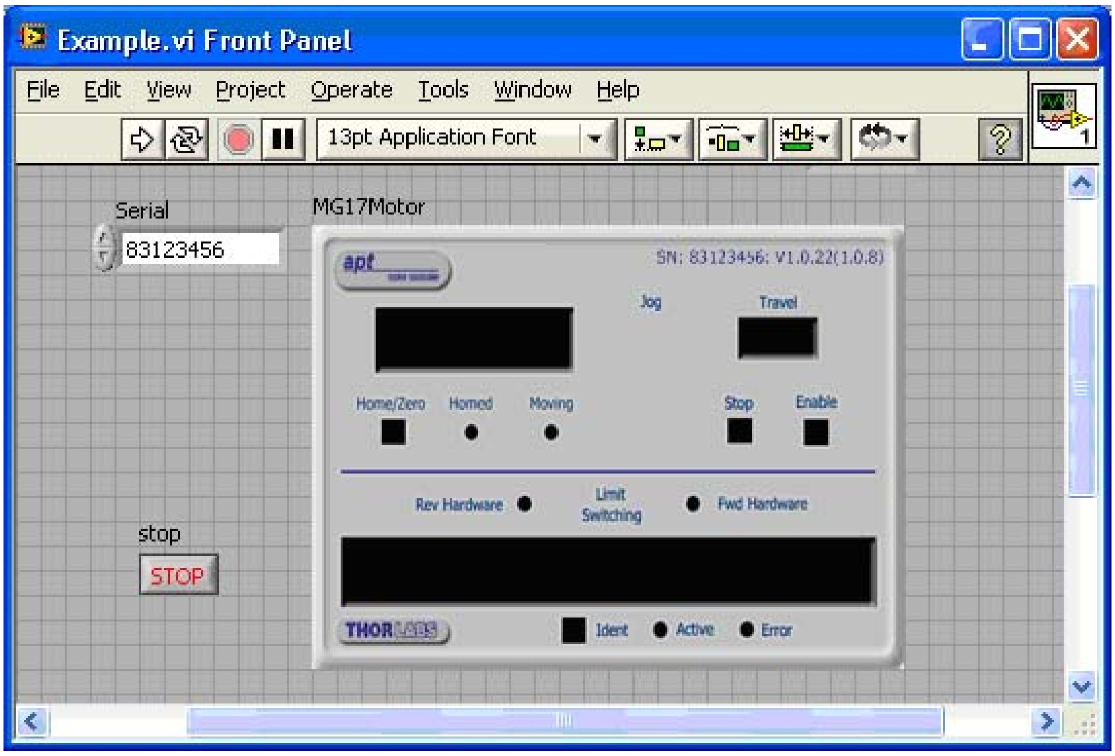


Figure 2-1 – Example Front Panel

You build the front panel with controls and indicators, which forms the user interface. Controls are knobs, push buttons, dials, and other input devices. Indicators are numerical readouts, LEDs, and other displays. Controls simulate the physical interface of an instrument and supply data to the block diagram of the VI. Indicators simulate instrument displays and are used to display data from the block diagram.

Block Diagram

After you build the front panel, you add code using graphical representations of functions to control the front panel objects. The block diagram contains this graphical source code. Front panel objects appear as terminals on the block diagram.

The VI in Figure 2-2 shows several block diagram objects - terminals, functions, and wires.

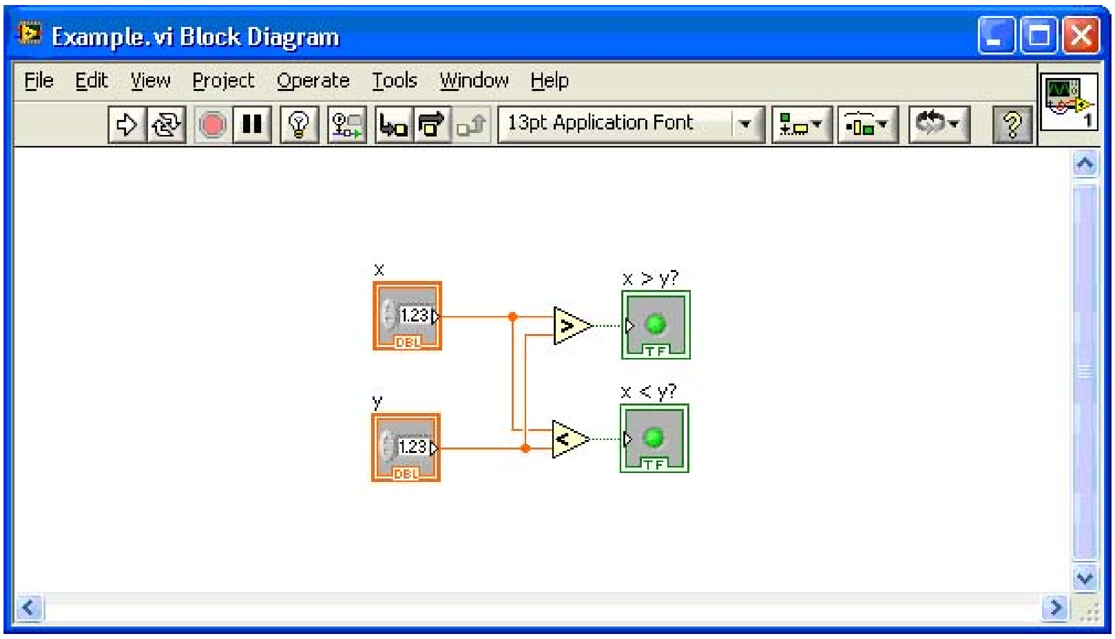


Figure 2-2 – Example of a Block Diagram

Terminals

The terminals represent the data of a control or indicator. You can configure front panel controls or indicators to appear as icons or data type terminals within the block diagram. By default, front panel objects appear as icon terminals. For example, the terminal shown top left (x), represents a numerical input control on the front panel. The DBL at the bottom of the terminal represents a data type of double-precision, floating-point numeric.

Terminals are entry and exit ports that exchange information between the front panel and block diagram. Data you enter into the front panel controls (e.g. **x** and **y** in Figure 2-2) enter the block diagram through the control terminals.

In the example here the data then enters the **Greater** and **Less** functions. When the functions complete their internal calculations, they produce new data values and in this case types. The data flows to the indicator terminals, where they exit the block diagram, re-enter the front panel, and appear on front panel indicators (**x>y** and **x<y** in Figure 2-3).

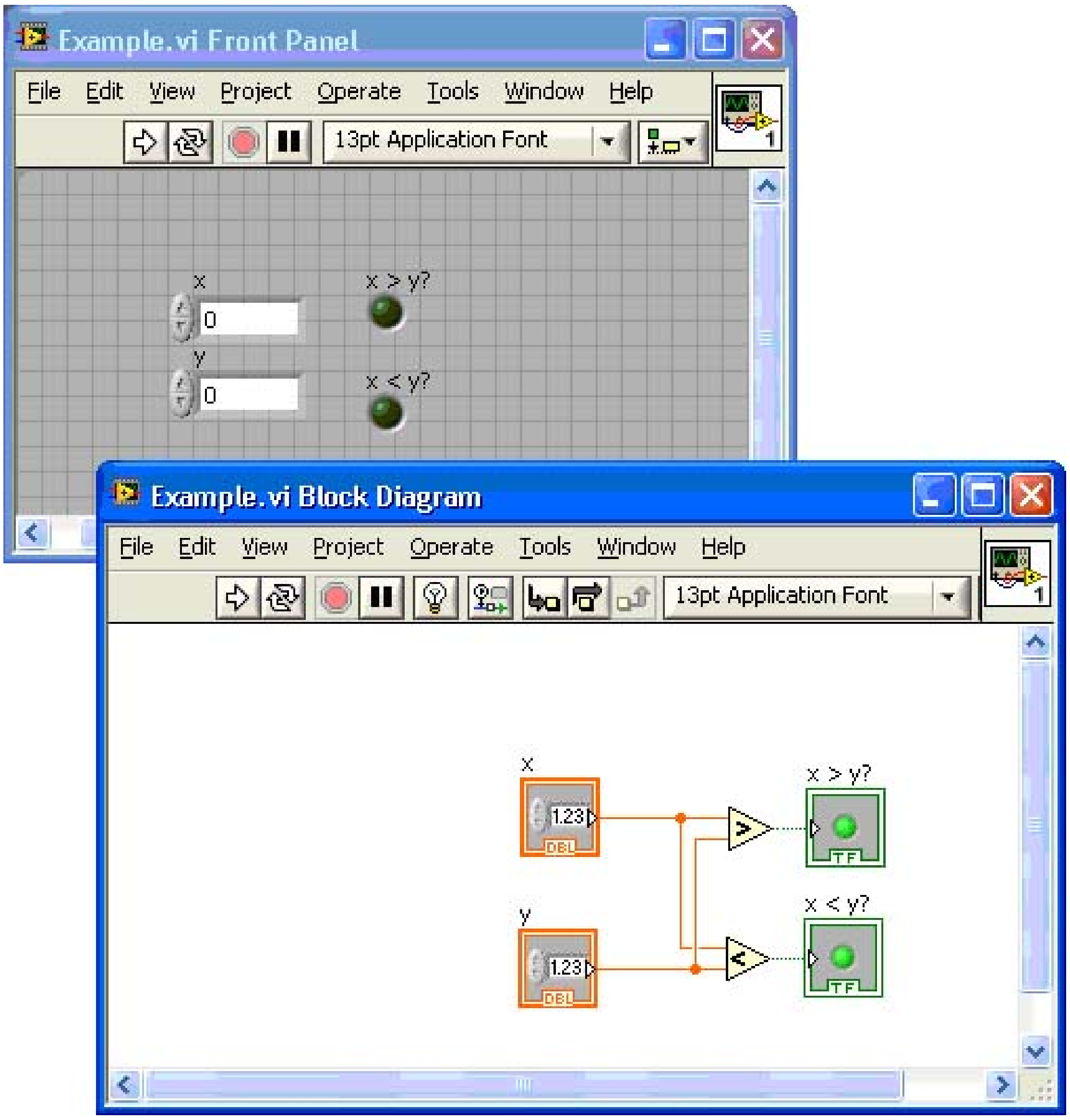


Figure 2-3 – Front Panel and Block Diagram Example

Nodes

Nodes are objects in the block diagram that have inputs and or outputs and perform operations when a VI runs. They are analogous to statements, operators, functions, and subroutines in text-based programming languages. The **Greater** and **Lower** functions in Figure 2-2 are examples of such nodes.

Wires

You transfer data between block diagram objects through wires. In Figure 2-2, wires connect the control and indicator terminals to the Greater and Less functions. Each wire has a single data source, but you can split a wire to supply many VIs and functions that read the data. Wires are different colours, styles, and thicknesses, depending on their data types. A broken wire appears as a dashed black line with a red X in the middle. Broken wires prevent a VI from running and must be removed.

Structures

Structures are graphical representations and are analogous to loops and case statements of text-based programming languages. Use structures in the block diagram to repeat blocks of code and to execute code conditionally or in a specific order.

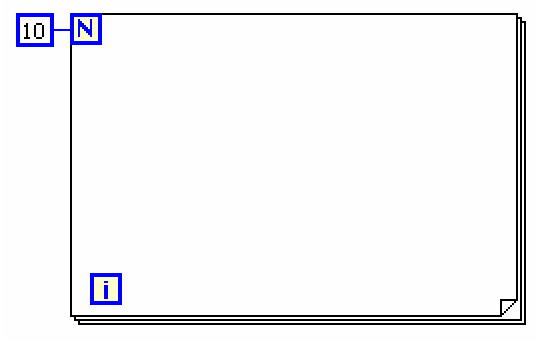


Figure 2-4 – Example For Structure

Icon and Connector Pane

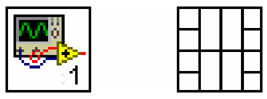


Figure 2-5 – VI Icon and Connector Pattern

After you build a VI front panel and block diagram, you can build the icon and the connector pane so you can use the VI as a subVI. Every VI displays an icon, such as the one shown in Figure 2-5,, in the upper right corner of the front panel and block diagram windows.

An icon is a graphical representation of a VI. If you use a VI as a subVI, the icon identifies the subVI on the block diagram of the VI. You can double-click the icon in the upper right corner of a VI to customize or edit it.

You may also need to build a connector pane, also shown in Figure 2-5, to use the VI as a subVI. The connector pane is a set of terminals that correspond to the controls and indicators of that VI, similar to the parameter list of a function call in text-based programming languages.

The connector pane defines the inputs and outputs you can wire to the VI so you can use it as a subVI. A connector pane receives data at its input terminals and passes the data to the block diagram code through the front panel controls and receives the results at its output terminals from the front panel indicators.

When you view the connector pane for the first time, you see a connector pattern. You can select a different pattern if you want to. The connector pane generally has one terminal for each control or indicator on the front panel. You can assign up to a maximum of 28 terminals to a connector pane. If you anticipate changes to the VI that would require a new input or output, leave extra terminals unassigned.

To view the connector right click the icon and select **Show Connector**. In a similar way to show the Icon whilst displaying the connector view right click the connector and select **Show icon**.

Chapter 3 The LabVIEW environment

Controls Palette

The **Controls** palette is available only on the front panel. The **Controls** palette contains the objects you use to create the front panel. The controls palette is organised into sub palettes based on their type.

The controls and indicators located on the **Controls** palette depend on the palette view currently selected.

Select **Window / Show Controls Palette** or right-click the front panel workspace to display the **Controls** palette. You can place the **Controls** palette anywhere on the screen. LabVIEW retains the **Controls** palette position and size so when you restart LabVIEW, the palette appears in the same position and has the same size.

Functions Palette

The **Functions** palette is available only on the block diagram. The **Functions** palette contains the VIs and functions you use to build the block diagram. The VIs and functions are located on sub palettes based on their type.
The VIs and functions located on the **Functions** palette depend on the palette view currently selected. Select **Window / Show Functions Palette** or right-click the block diagram workspace to display the **Functions** palette. You can place the **Functions** palette anywhere on the screen. LabVIEW retains the **Functions** palette position and size so when you restart LabVIEW, the palette appears in the same position and has the same size.

Context Help Window

The **Context Help** window displays basic information about LabVIEW objects when you move the cursor over each object. Objects with context help information include VIs, functions, constants, structures, palettes, properties, methods, events, and dialog box components. You also can use the **Context Help** window to determine exactly where to connect wires to a VI or function.

Select **Help /Show Context Help** to display the **Context Help** window.

You can place the **Context Help** window anywhere on the screen. The **Context Help** window resizes to accommodate each object description.
You also can resize the **Context Help** window to set its maximum size. LabVIEW retains the **Context Help** window position and size so when you restart LabVIEW, the window appears in the same position and has the same maximum size.
You can lock the current contents of the **Context Help** window so the contents of the window do not change when you move the cursor over
different objects. Select **Help / Lock Context Help** to lock or unlock the current contents of the **Context Help** window.

If a corresponding *LabVIEW Help* topic exists for an object the **Context Help** window displays, a blue **Click here for more help** link to extra information.

Chapter 4 Building the Front Panel

The front panel is the user interface of a VI. Generally, you design the front panel first, then design the block diagram to perform tasks on the inputs and outputs you create on the front panel.

Adding Objects to the Front Panel

You build the front panel with controls and indicators, which are the interactive input and output terminals of the VI, respectively. Controls are knobs, push buttons, dials, and other input devices. Indicators are graphs, LEDs, and other displays.

Select **Window / Show Controls Palette** to display the **Controls** palette, then select controls and indicators from the **Controls** palette and place them on the front panel.

Configuring Front Panel Objects

Use property dialog boxes or shortcut menus to configure how controls and indicators appear or behave on the front panel. Use property dialog boxes when you want to configure a front panel control or indicator through a dialog box that includes context help or when you want to set several properties at once for an object. Use shortcut menus to quickly configure common control and indicator properties. Options in the property dialog boxes and shortcut menus differ depending on the front panel object. Any option you set using a shortcut menu overrides the option you set with a property dialog box.

Right-click a control or indicator on the front panel and select **Properties** from the shortcut menu to access the property dialog box for that object.

Note: You cannot access property dialog boxes for a control or indicator while a VI runs.

Showing and Hiding Optional Elements

Front panel controls and indicators have optional elements you can show or hide. Set the visible elements for the control or indicator on the **Appearance** tab of the property dialog box for the front panel object. You also can set the visible elements by right-clicking an object and selecting **Visible Items** from the shortcut menu. Most objects have a label and a caption that can be shown or hidden.

Changing Controls to Indicators and Indicators to Controls

LabVIEW initially configures objects in the **Controls** palette as controls or indicators based on their typical use. For example, if you select a toggle switch it appears on the front panel as a control because a toggle switch is usually an input device. If you select an LED, it appears on the front panel as an indicator because an LED is usually an output device.

Some palettes contain a control and an indicator for the same type or class of object. For example, the **Numeric** palette contains a digital control and a digital indicator.

You can change a control to an indicator by right-clicking the object and selecting **Change to Indicator** from the shortcut menu, and you can change an indicator to a control by right-clicking selecting **Change to Control**.

Chapter 5 Building the Block Diagram

After you build the front panel, you add code using graphical representations of functions to control the front panel objects. The block diagram contains this graphical source code.

The relationship between Front Panel Objects and Block Diagram Terminals

For each front panel object a corresponding block diagram terminal is created. Double-click a block diagram terminal to highlight the corresponding control or indicator on the front panel.

Terminals are entry and exit ports that exchange information between the front panel and block diagram. Data you enter into the front panel controls enter the block diagram through the control terminals. During execution, the output data flows to the indicator terminals, where they exit the block diagram, re-enter the front panel, and appear in front panel indicators.

Block Diagram Objects

Objects in the block diagram include terminals, nodes, and functions.
You build block diagrams by connecting the objects with wires.

Block Diagram Terminals

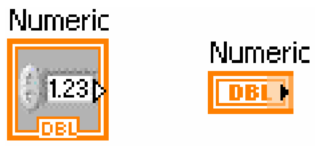


Figure 5-1 – Icon & Data Type View

You can configure front panel controls or indicators to appear as icons or data type terminals on the block diagram. By default, front panel objects appear as icon terminals.

For example, a numeric icon terminal, shown in Figure 5-1, represents a numeric input on the front panel. The DBL at the bottom of the terminal represents a data type of double-precision, floating-point numeric.
The DBL terminal, also shown in Figure 5-1, represents the same control in the data type view, where DBL represents the data type.

Right-click a terminal and select **Display Icon** from the shortcut menu to remove the checkmark and to display the terminal in the data type view. Use icon terminals to display the types of front panel objects on the block diagram, in addition to the data types of the front panel objects. Use data type terminals to conserve space on the block diagram.

Note Icon terminals are larger than data type terminals, so you might unintentionally obscure other block diagram objects when you convert a data type terminal to an icon terminal.

A terminal is any point to which you can attach a wire, other than to another wire. LabVIEW has control and indicator terminals, constants, and other specialized terminals. You use wires to connect terminals and pass data to other terminals.

Right-click a block diagram object and select **Visible Item / Terminals** from the shortcut menu to view the terminals. Right-click the object and select **Visible Item / Terminals** again to hide the terminals.

Note: This shortcut menu item is not available for expandable VIs and functions.

Block Diagram Nodes

Nodes are objects on the block diagram that have inputs and/or outputs and perform operations when a VI runs. They are analogous to statements, operators, functions, and subroutines in text-based programming languages.

Using Wires to Link Block Diagram Objects

You transfer data among block diagram objects through wires. Each wire has a single data source, but you can wire it to many VIs and functions that read the data. You must wire all required block diagram terminals, otherwise the VI is broken and will not run.

Display the **Context Help** window to see which terminals a block diagram node requires. The labels of required terminals appear bold in the **Context Help** window.

Wires are different colours, styles, and thicknesses depending on their data types. A broken wire appears as a dashed black line with a red X in the middle. The arrows on either side of the red X indicate the direction of the data flow, and the colour of the arrows indicate the data type of the data flowing through the wire.

Wire stubs are the truncated wires that appear next to unwired terminals when you move the wiring tool over a VI or function node. They indicate the data type of each terminal. A tool tip also appears, listing the name of the terminal. After you wire a terminal, the wire stub for that terminal does not appear when you move the Wiring tool over its node.

A wire segment is a single horizontal or vertical piece of wire. A bend in a wire is where two segments join. The point at which two or more wire segments join is a junction. A wire branch contains all the wire segments from junction to junction, terminal to junction, or terminal to terminal if there are no junctions in between.

While you are wiring a terminal, bend the wire at a 90 degree angle once by moving the cursor in either a vertical or horizontal direction. To bend a wire in multiple directions, click the mouse button to set the wire and then move the cursor in the new direction. You can repeatedly set the wire and move it in new directions.

To undo the last point where you set the wire, press the <Shift> key and click anywhere on the block diagram.

When you cross wires, a small gap appears in the first wire you drew to indicate that the first wire is under the second wire.

TIP: Crossing wires can clutter a block diagram and make the block diagram difficult to debug. Avoid crossing wires as often as possible.

Chapter 6 Getting Started

Building a VI with APT

The **Getting Started** window, shown in Figure 6-1, appears when you launch LabVIEW. Use this window to create new VIs, select among the most recently opened files, and to find examples.

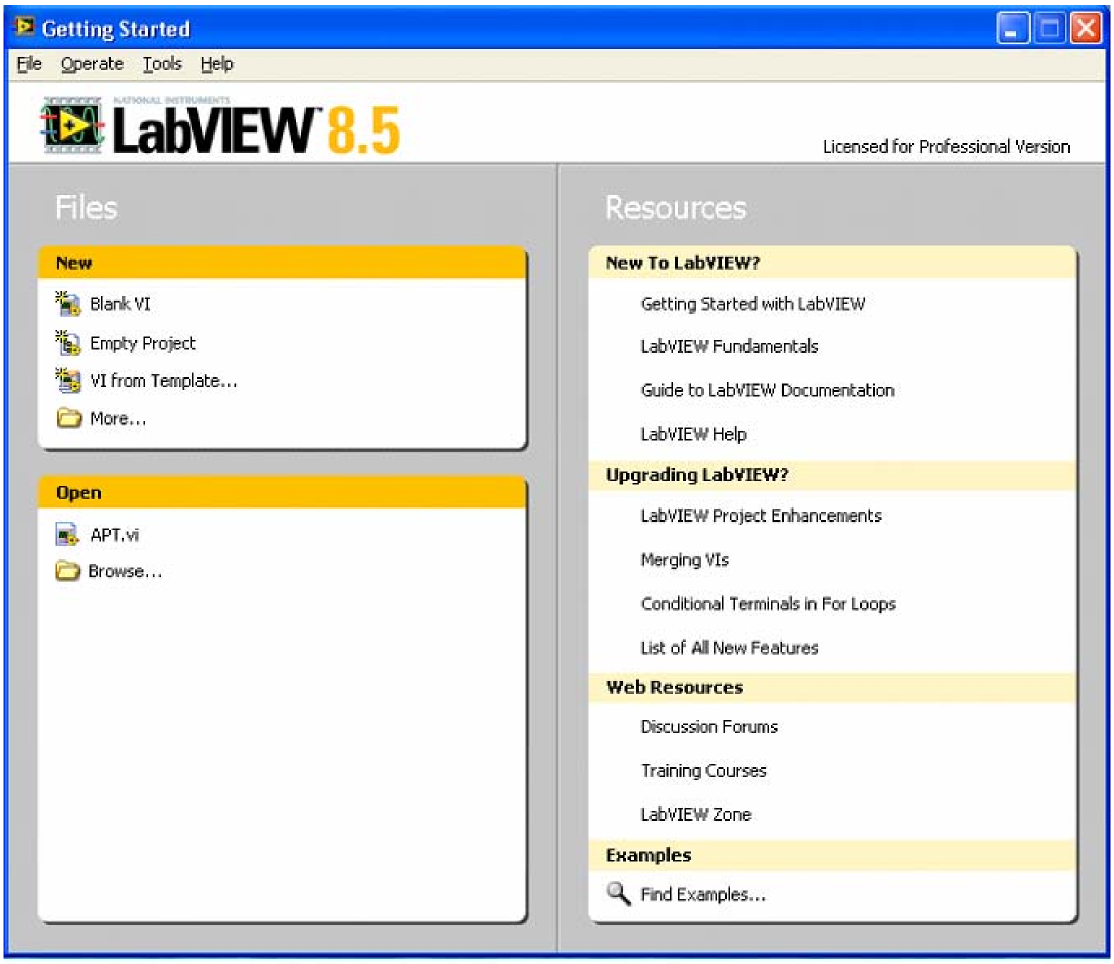


Figure 6-1 – Getting started Screen

In the **Getting Started** window, click the **New / Blank VI** link to create a new blank VI. LabVIEW displays two windows: the front panel window and the block diagram window.

- The front panel, or user interface, appears with a gray background and is used to display controls and indicators. The title bar indicates that this window is the front panel for the blank VI.
- The block diagram appears with a white background and includes sub Vis and graphical code structures that control the front panel objects. The title bar indicates that this window is the block diagram for the blank VI.

Note: If the front panel is not visible, you can display the front panel by selecting **Window / Show Front Panel**. Alternatively if the front panel is not visible, you can display the block diagram by selecting **Window / Show Block Diagram**.

Adding Controls to the Front Panel

Controls on the front panel provide a way to input and output data through the use of controls and indictors to the block diagram of the VI.

Complete the following steps to add a serial number entry and stop button control to the front panel.

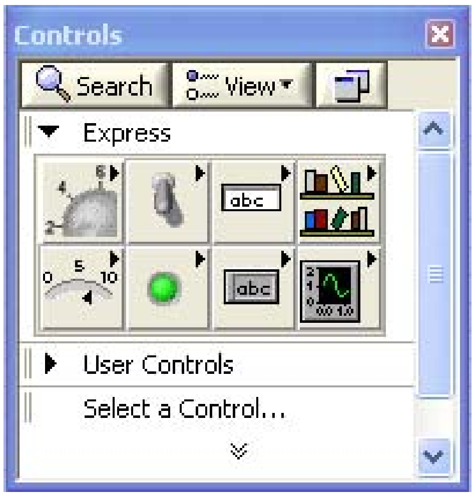


Figure 6-2 – Controls Palette

1) If the **Controls** palette, shown in Figure 6-2, is not visible on the front panel, select **View / Controls Palette**.

2) The **Controls** palette opens with the **Express** sub palette visible by default. If you do not see the **Express** sub palette, click **Express** on the **Controls** palette to display the **Express** sub palette.

3) Click the **Numeric Controls** icon to display the **Numeric Controls** palette.

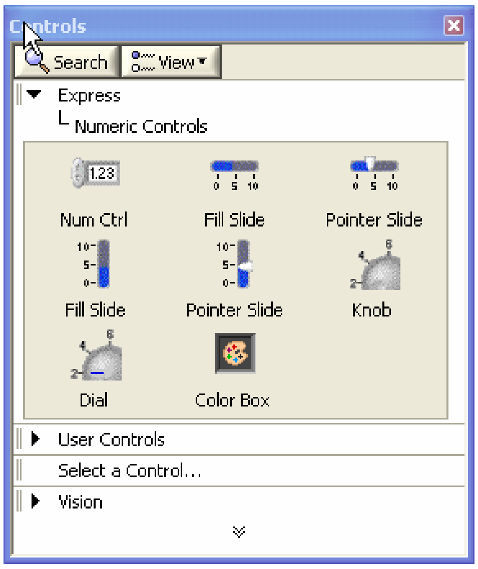
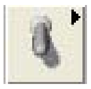


Figure 6-3 – Numeric Controls Palette

4) Click the **Numeric Control** to attach it to the cursor, and then click in the Front Panel to add the control.

5) Double click the default name given “**Numeric**” and change it to “**Serial**”. Adjust the size of the control to allow sufficient space to input the 8 digit serial number of your APT controller.

6) Click the Buttons & Switches Icon  to display the **Buttons & Switches** palette.

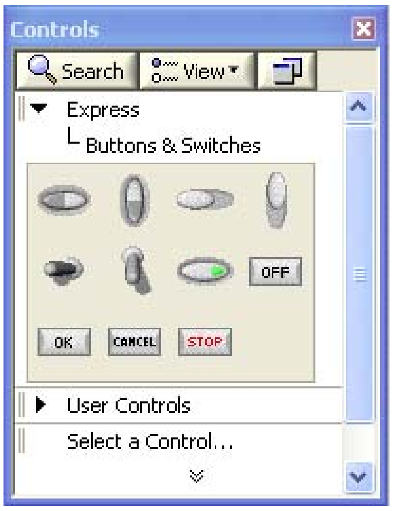


Figure 6-4 – Buttons & Switches Palette

7) Click the STOP Button icon to select it, then click in the Front Panel to add the control.

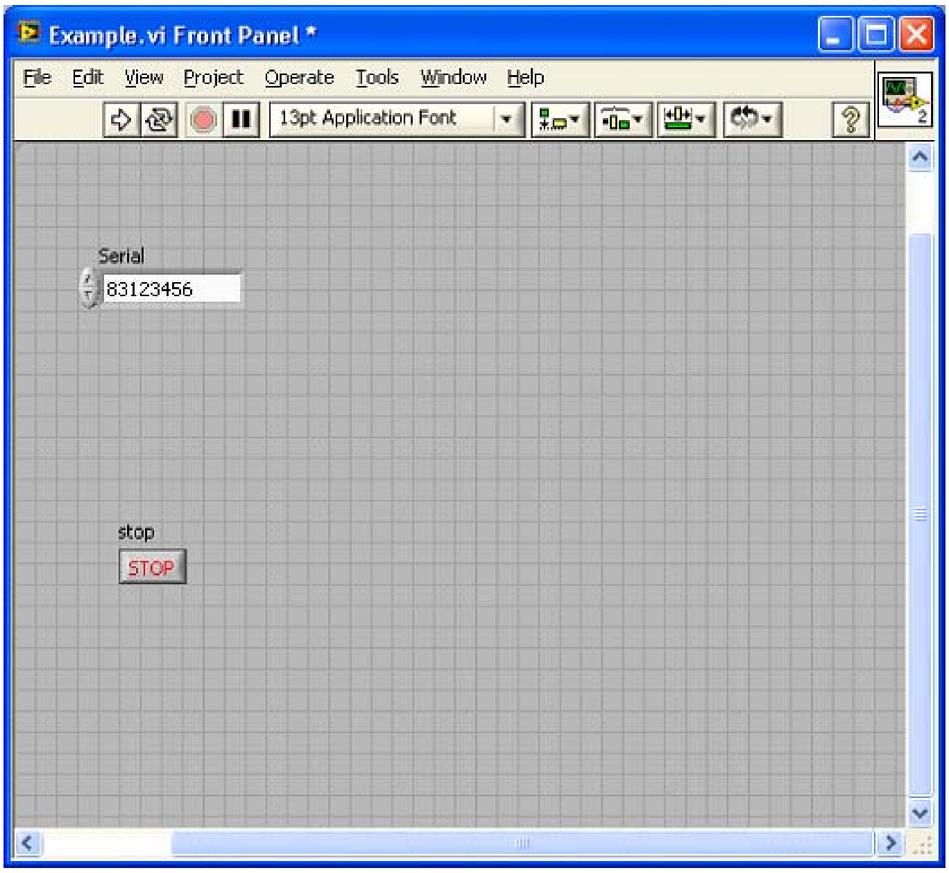


Figure 6-5 – VI with Numeric and Stop Control

It is a good idea at this stage to set a default value for the serial number to be used, as this will always be the same if only one piece of equipment is being used.

8) Enter your 8 digit APT motor controller serial number into the numeric control. In this example the serial number used is 83123456.

9) Right click the numeric control and select from the shortcut menu **Data Operations / Make Current Value Default**. This stores the entered value as a default value.

10) Select **File / Save As** and save the VI as **Example.vi** in an easily accessible location. This allows the VI and subsequent changes to be saved without danger of losing the work carried out.

Adding an ActiveX control to the front panel

LabVIEW provides many standard controls and also has the ability to host third party controls through mechanisms such as ActiveX.

APT software is exposed through ActiveX to allow users to incorporate hardware control through their own custom applications.

Complete the following steps to add an APT Motor Control to the front panel.

1) Click the double chevron to expand the **Controls** palette, then select the **.NET & ActiveX** palette. If the Controls palette is not visible, select **View/Controls Palette.**

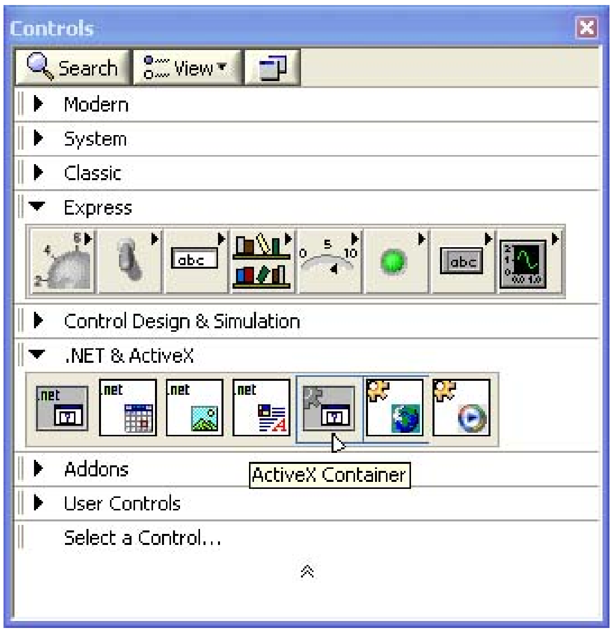


Figure 6-6 – Controls Palette

2) Select the **ActiveX Container** to attach it to the cursor, and then place the control on the front panel. Notice at this stage the container is empty.

3) Right click on the centre of the ActiveX Container and select from the shortcut menu **Insert ActiveX Object…**

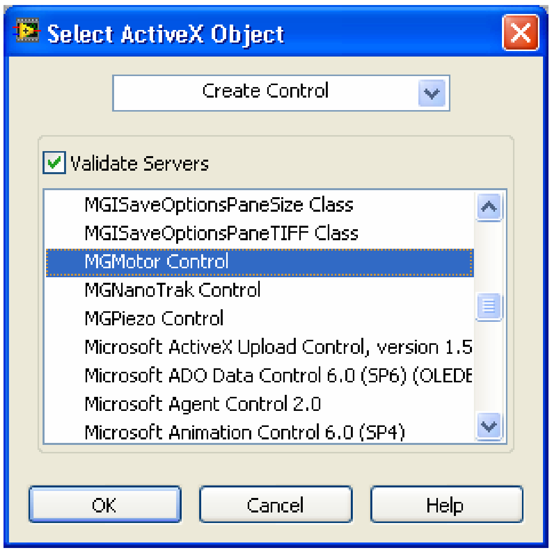

Select ActiveX Object
Create Control
Validate Servers
MGISaveOptionsPaneSize Class
MGISaveOptionsPaneTIFF Class
MGMotor Control
MGNanoTrak Control
MGPiezo Control
Microsoft ActiveX Upload Control, version 1.5
Microsoft ADO Data Control 6.0 (SP6) (OLEDE
Microsoft Agent Control 2.0
Microsoft Animation Control 6.0 (SP4)
OK
Cancel
Help

Figure 6-7 – ActiveX Object Selector Window

4) From the list view of creatable objects select **MGMotor Control**, then click OK.

Note: In LabVIEW the **MGMotor** control represents the ActiveX control used to interface with motor controller type hardware. Other APT hardware types have their own ActiveX controls. Refer to the APT programming guide for further information on control names used with specific types of hardware.

5) The ActiveX container should now contain the APT motor control. Resize and position the control as required as shown in Figure 6-8.

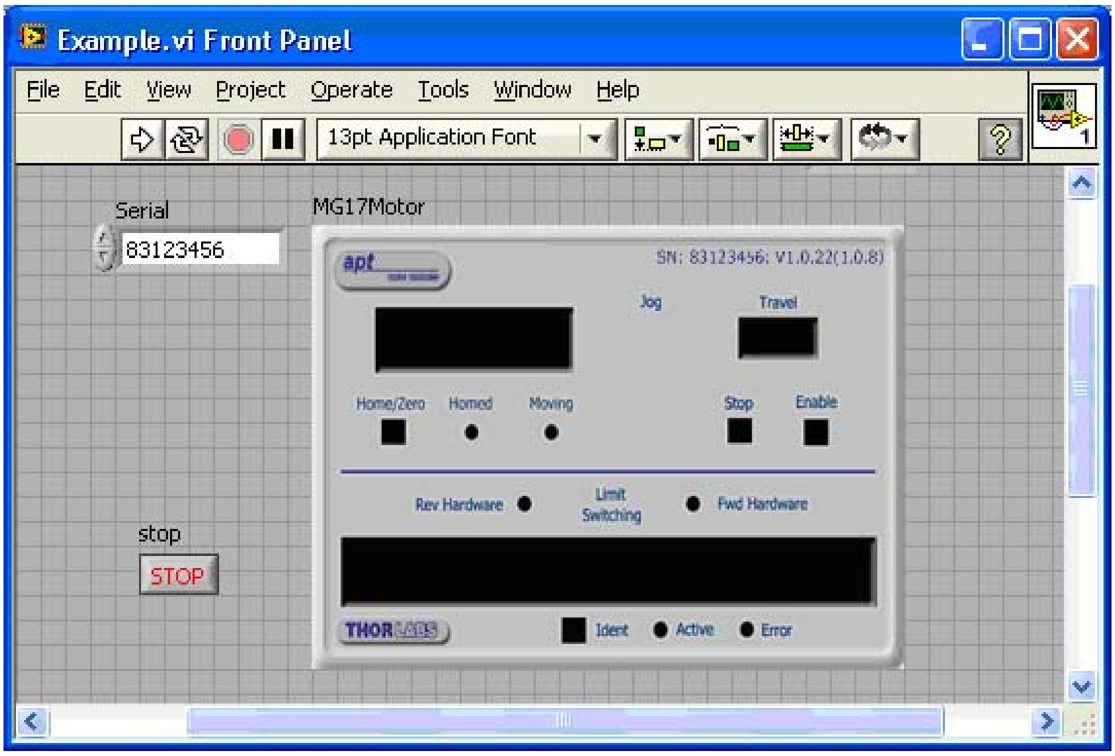


Figure 6-8 – ActiveX container with APT Motor Control

6) Select **File/Save** to save the changes.

Wiring Objects on the Block Diagram

To make use of the controls on the front panel, you must interconnect the objects with graphical code within the block diagram. For each front panel object there is a corresponding block diagram terminal. In this example there are three such terminals, as shown in Figure 6-9.

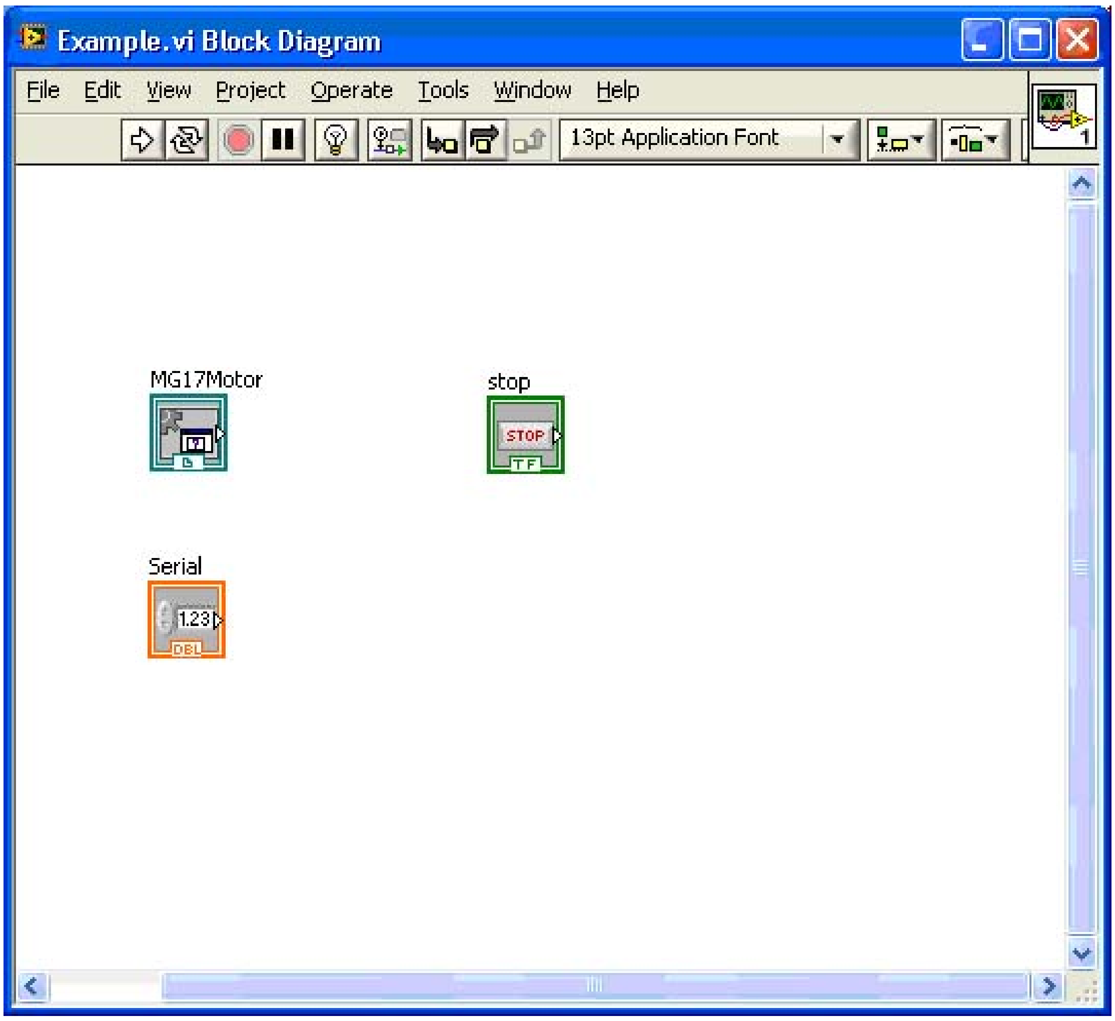


Figure 6-9 – Block Diagram Terminal Node

Complete the following steps to wire the stop button to a while loop.

1) If the block diagram is not visible select from the front panel menu bar **Window / Show Block Diagram**.

2) If the **Functions** palette, shown in Figure 6-10 is not visible on the front panel, select **View / Functions Palette**, alternatively right click the block diagram workspace.

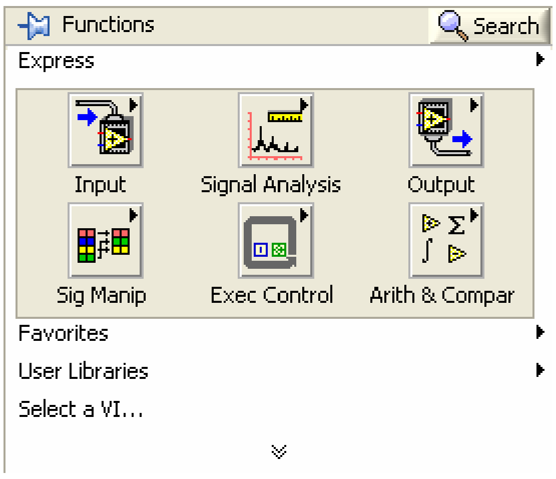


Figure 6-10 – Functions Palette

3) The **Functions** palette opens with the **Express** sub palette visible by default. Click the double chevron to expand the list of palettes available then select **Programming/Structure** to display the **Structures palette** shown in Figure 6-11.

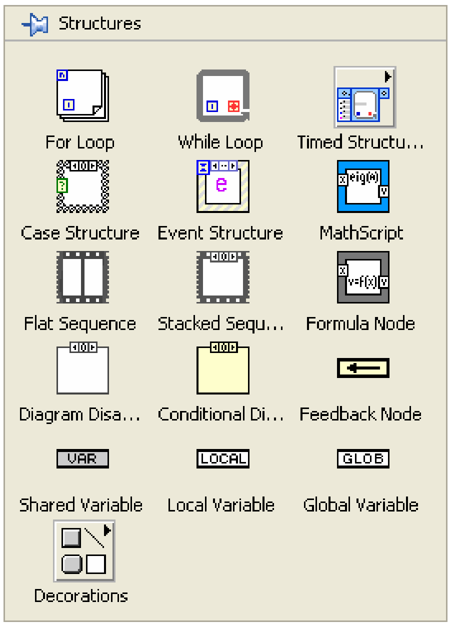


Figure 6-11 – Structures Palette

4) Select the **While Loop** icon to attach it to the cursor. And then draw a rectangle round the stop button as in the block diagram shown in Figure 6-12.

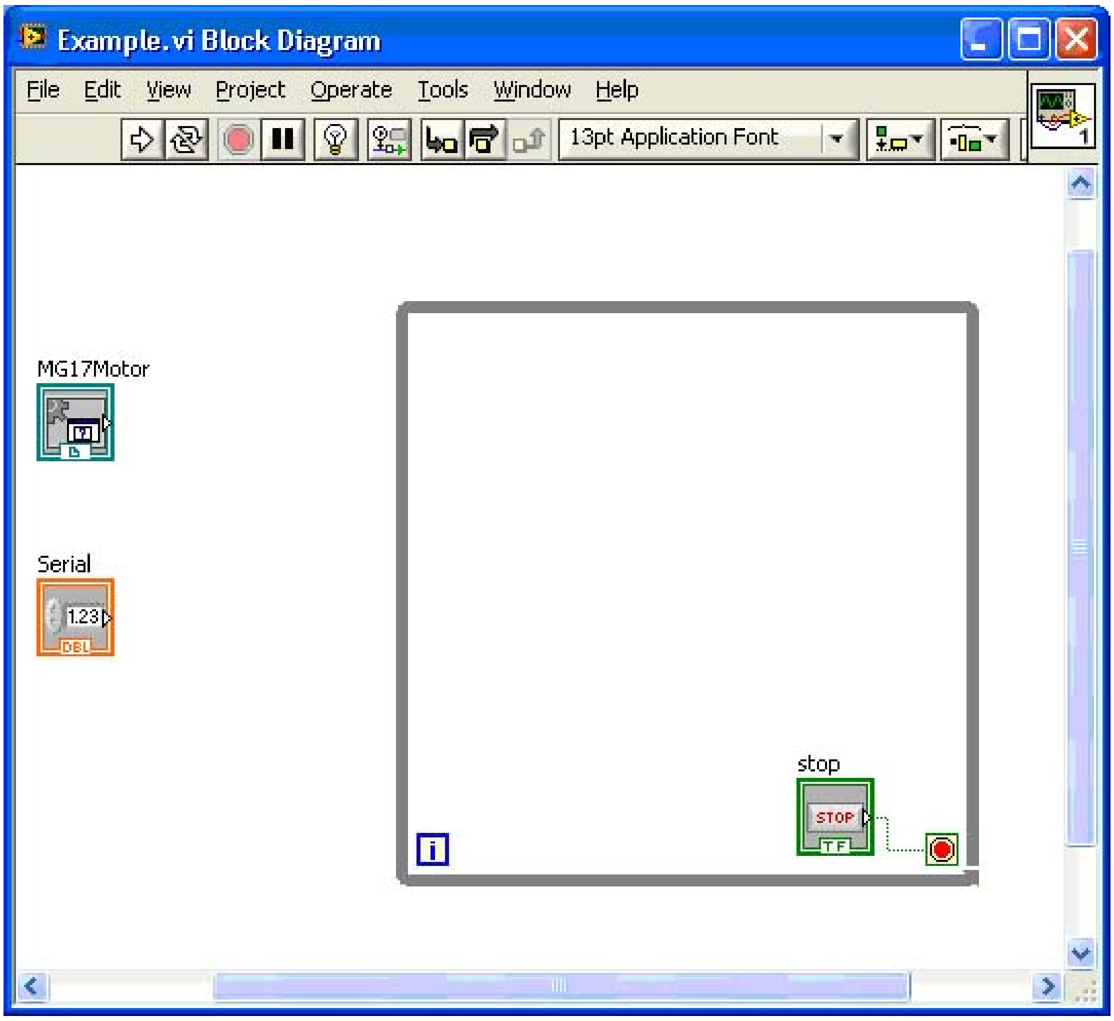


Figure 6-12 – While Loop around stop button

5) Click on the **stop** terminal output connector to start wiring, and then click on the **while loop stop button** to terminate wiring. This joins the stop button action to the while loop stop logic.
Note. Rearrange/resize the wiring diagram objects as necessary whilst creating this example.
6) Select File/Save to save the changes.

Tip: Should you find that wiring cannot be started select **View / Tools Palette** and enable automatic tools selection as shown in Figure 6-13.

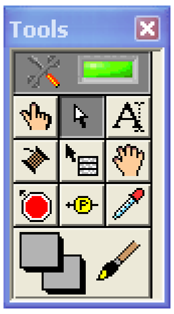


Figure 6-13 – Block Diagram Tools Palette with Automatic Tool Enabled

Setting an ActiveX property

Properties of the APT control objects can be set through code in the block diagram.

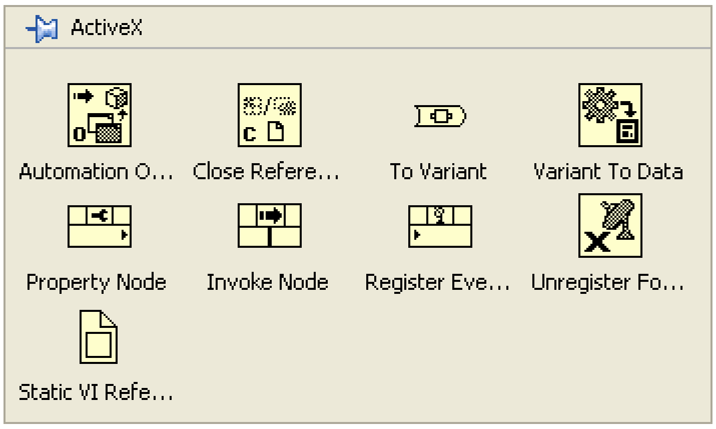


Figure 6-14 – ActiveX Palette

Complete the following steps to set the serial number property of the APT motor control object.

1) Select the Block Diagram window. If not visible, select **Window/Show Block Diagram**.

2) Display the ActiveX palette as shown in Figure 6-14 by right clicking the block diagram workspace to display the functions palette and then select **Functions Palette/Connectivity/ActiveX**

3) Select the **Property Node** icon to attach it to the cursor. Then drop the node onto the block diagram.

4) Wire the APT control object terminal in the block diagram to the **input reference** connector of the property node by first clicking on the APT control output connection to begin wiring. Click on the **input ref** connection of the property node to complete the wiring.

5) In wiring the property node to the APT control object terminal the property node now has information available as to what properties are available.

6) Right click the white area of the property node which by default displays the word **property**. In the shortcut menu select **properties**. The available properties are presented in a further shortcut drop down menu. Select the **HWSerialNum** property from the list.

7) The property node now automatically displays the correct property node name and connectors.

8) By default the property node parameter is set to read the current value. In this instance we would like to set the serial number. Right click the white area of the property node again and select **Change All To Write** from the shortcut menu that is presented.

9) To set the property, data needs to be connected to the input of the property node. Wire the input of the property node to the serial number control terminal by first clicking the serial number terminal output connection to begin wiring. Click on the property node **HWSerialNum** input connection to complete the wiring.

Notice that the wire between the serial number numeric and the property node has a dot on the end closest to the property node. This is indicating that a data type conversion is taking place as the numeric data type is a double floating point number and the APT serial number property is a 32 bit integer. It is appropriate at this stage to change the data type representation of the front panel serial number input control to an integer, to avoid this unnecessary overhead.

10) Right click the **Serial** terminal in the block diagram. From the shortcut menu displayed select **Representation/I32**. This changes the numeric control to be represented by a 32 bit integer.

Notice that the dot on the wire input of the property node has now been removed as there is no need for any data type conversion. Also, the Serial terminal changes colour from orange to blue.

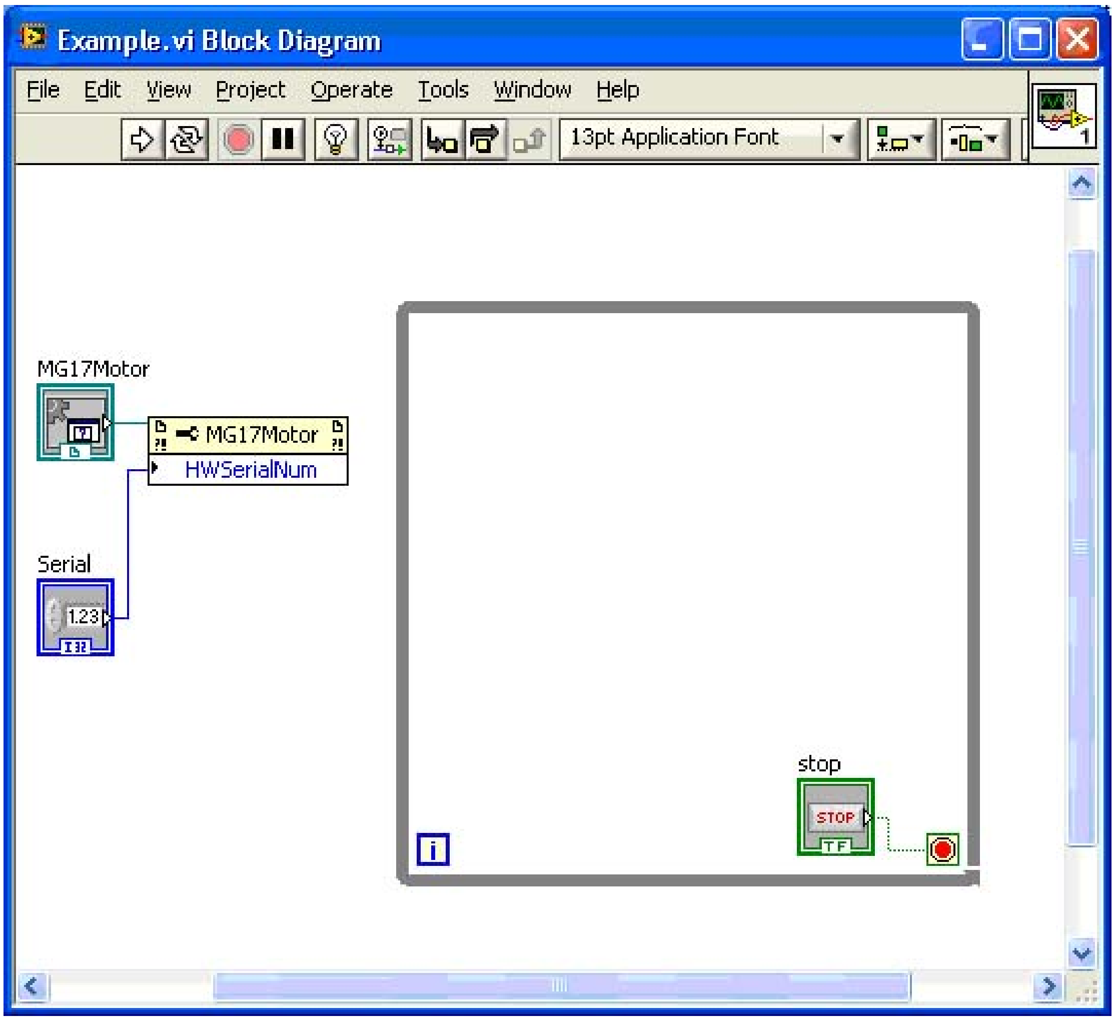


Figure 6-15 – Property Node Wired to ActiveX Control

11) Select **File/Save**, to save the changes.

Calling an ActiveX Method

Methods of the APT control object can be called within the block diagram by using an **Invoke Node**. Details and information about specific APT methods can be found in the programming help file of the APT software.
In this example we will call the **StartCtrl** (Start Control) method which starts communication with connected APT hardware.

Complete the following steps to call the Start Control method of the APT control object.

1) Select the Block Diagram window. If not visible, select **Window/Show Block Diagram**.

2) Display the ActiveX palette by right clicking the block diagram workspace to display the functions palette and then navigate to **Functions Palette/Connectivity/ActiveX**.

3) Select the **Invoke Node** icon to attach it to the cursor. Then drop the node onto the block diagram.

To ensure that the serial number is set prior to calling the start control method we wire the output reference of the previous **HWSerialNum** property node to the input reference of the **StartCtrl** invoke node. This reference is the same reference as the original APT control object, however using the output side of the property node ensures that the serial number will be set before calling the start control method.

4) Click the **reference output** of the **property node** to begin wiring. Click the **reference input** of the **invoke node** to complete the wiring.

In wiring the invoke node to the APT control object reference the invoke node now has information available as to what methods are available.

5) Right click the white area of the invoke node which by default displays the word **method**. In the shortcut menu displayed select **Methods**. The available methods are presented in a further shortcut menu. Select the **StartCtrl** method from the list.

The invoke node now automatically displays the correct method name.
This particular method requires no further parameters. However, there is a returning parameter which for the time being we will leave this unwired.

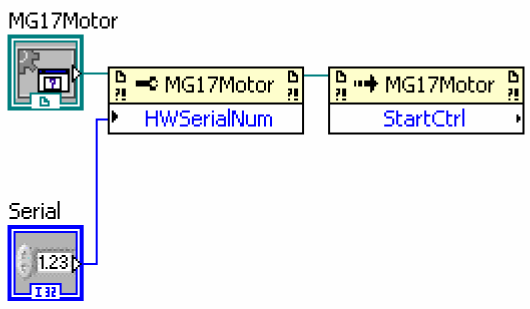


Figure 6-16 – Invoke Node Wired to ActiveX control

In a similar fashion we need to wire a second method, **StopCtrl,** as in Figure 6-17. This complements the start control method to terminate communication prior to the program finishing.

6) Display the ActiveX palette by right clicking the block diagram workspace to display the functions palette and then navigate to **Functions Palette / Connectivity / ActiveX**.

7) Select the **Invoke Node** icon to attach it to the cursor. Then drop the node onto the block diagram.

8) Click the **reference output** connector of the **StartCtrl** invoke node to begin wiring. Click the left hand edge of the **while loop** to complete the wiring segment.

9) Click the connector block just created on the left hand edge of the **while loop** to begin wiring again. Click the right hand edge of the **while loop** to complete the wiring segment.

10) Click the connector block just created on the right hand edge of the **while loop** to begin wiring again. Click the **reference** input connector on the second **invoke node**.

11) Right click the white area of the second invoke node which by default displays the word **method**. In the shortcut menu displayed select **Methods**. The available methods are presented in a further shortcut menu. Select the **StopCtrl** method from the list.

Note: The reference of the second **invoke node** enters and exits the **while loop**. Because the **while loop** does not terminate until the stop button is pressed, the second **invoke node** inputs cannot be satisfied and therefore will not execute until the loop is exited.

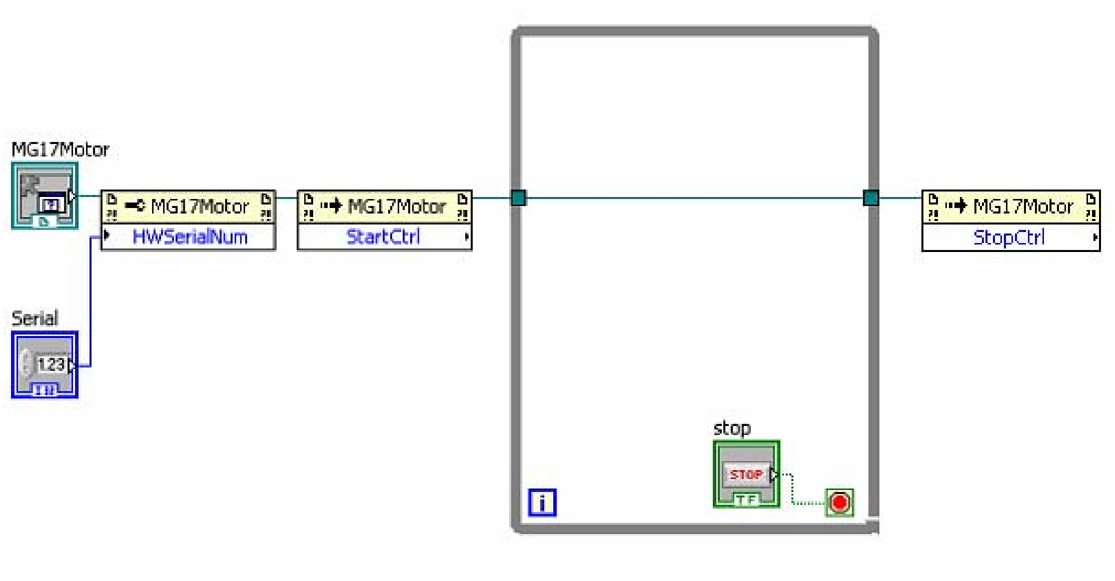


Figure 6-17 – Second Invoke Method wired to ActiveX control

12) Re-arrange the block diagram objects a shown in Figure 6.14 as necessary.

13) Click **File/Save** to save the changes

Passing parameters to an ActiveX method

ActiveX methods can have both input and output parameters and a return parameter. An example of one such APT motor method, **GetPosition**, is shown in Figure 6-18.

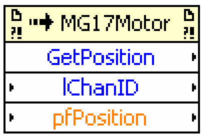


Figure 6-18 – APT ActiveX method with Input\Output Parameters

In this particular method there is one input parameter, one output parameter and a return parameter. For specific details regarding this method see the APT programming help file.

The input parameter **lChanID** is a data type passed in by value and subsequently needs to be provided with an integer in this case. A pass through value is available to subsequent code which is available on the output side of the parameter. This value is unchanged as it is used by value.

The output parameter **pfPosition** is a data type returned by reference and subsequently needs to be supplied with a data reference on both input and output. There are several techniques to do this. The easiest to understand when reviewing code is to provide the same variable as an input and as an output.

The return value is in this instance an integer, the value of which can simply be read and used.

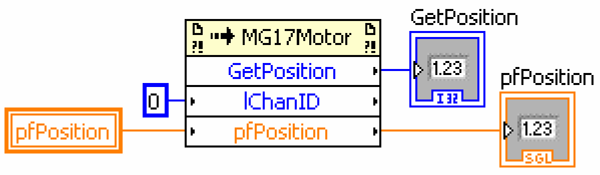


Figure 6-19 – ActiveX Method call with Inputs\Outputs and a Return Value

In the example shown in Figure 6-19, a local variable is used to provide the method parameter a reference to a variable to satisfy the input requirements. This variable is the same variable connected to the output of the parameter.

Effectively in Figure 6-19, reading from left to right, the method parameter is read, modified, returned and set.

Complete the following steps to call the **GetPosition** method of the APT control object.

1) Select the Block Diagram window. If not visible, select **Window/Show Block Diagram**.

2) Display the ActiveX palette by right clicking the block diagram workspace and displaying the functions palette and navigating to **Functions Palette/Connectivity/ActiveX**.

3) Select the **Invoke Node** icon to attach it to the cursor. Then drop the node onto the block diagram within the while loop.

4) Select the wiring between the start and stop method which resides within the **while loop**, press the **delete key** to remove the section of wire.

5) Click the connector block on the left hand edge of the **while loop** to begin wiring. Click the **reference input** connector of the new **invoke node** to complete the wiring.

6) Similarly attach a wire to the **reference output** connector of the same **invoke node** by clicking the connector to begin wiring. Click the connector block on the right hand edge of the **while loop** to complete the wiring.

7) Right click the white area of the new invoke node which by default displays the word **method**. In the shortcut menu displayed select **Methods**. The available methods are presented in a further shortcut menu. Select the **GetPosition** method from the list.

8) The **invoke node** now automatically displays the correct method name and parameters.

9) Right click the input connector of the parameter **lChanID** and select from the shortcut menu displayed **Create / Constant**. A suitable data type constant is created and wired automatically. The default value of zero is sufficient.

10) Right click the output connector of the parameter **pfPosition** and select from the shortcut menu displayed **Create / Indictor**. An indicator, of the correct data type, is created and wired automatically.

11) Right click the newly created indicator and select from the displayed shortcut menu **Create / Local Variable**. A local variable is created which represents the indicator. By default the variable is of a write type.

12) Right click the **local variable** and select from the shortcut menu **Change To Read**.

13) Wire the **local variable** to the input of the **pfPosition** parameter by clicking the output connector of the **local variable** to begin wiring, then click the pfPosition input connector of the new **invoke node** to complete the wiring.

14) Select **File/Save** to save the changes.

The resulting VI should look like that shown in Figure 6-20. Nodes and wiring can be repositioned to reflect the example documented for ease of understanding in further sections of this document.

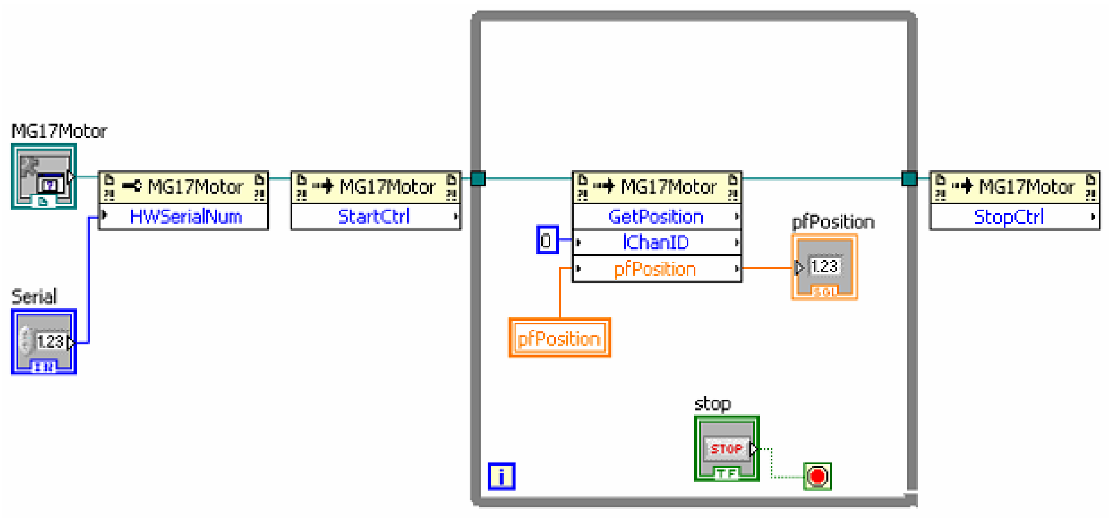


Figure 6-20 - ActiveX Method call which returns information from APT

Controlling Code Execution Speed and Flow

Controlling Speed

Although not completely necessary to ensure correct VI execution, it is good programming practice in LabView to limit code execution speed where possible. If we take the example that has been built in the previous sections the code will attempt to read the motor controller's position as fast as possible and display for the user.
This is a good example of where program execution speed can be reduced without affecting the program functionality. For a user displayed variable there is little to gain by updating at speeds of more than 5Hz, as the user would not notice the extra speed and may even be unreadable.
Within the while loop a short delay to hold execution by 200ms would ensure that the CPU and APT system are not over burdened with unnecessary processing. A time delay VI can be used to achieve this.

Complete the following steps to insert a wait call within the while loop to reduce the execution speed of the loop.

1) Select the Block Diagram window. If not visible, select **Window/Show Block Diagram**.
2)
3) Display the Timing palette by right clicking the block diagram to display the functions palette and then navigate to **Functions Palette/Programming/Timing**.

4) Select the **Wait (ms)** icon to attach it to the cursor. Then drop the VI onto the block diagram within the **while loop**.

5) Right click the input connector of the **Wait(ms)** VI and select **Create/Constant** from the shortcut menu displayed. A constant, of suitable data type, is created and wired automatically. Edit the default value by double clicking the constant and enter 200. The units are milliseconds.

6) Select **File/Save** to save the changes.

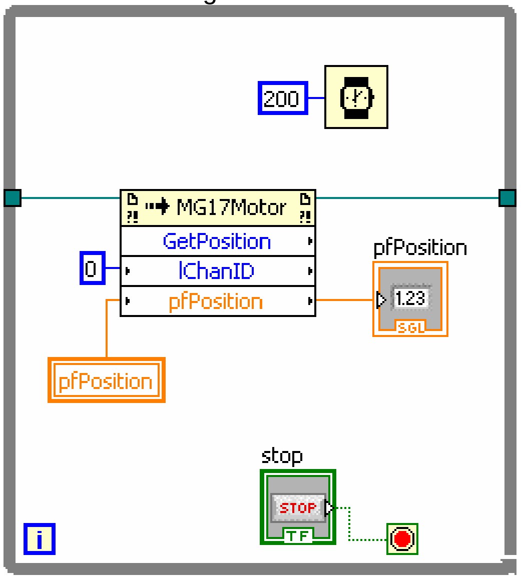


Figure 6-21 – While loop with controlled execution speed

For every iteration of the while loop a single hold of execution will occur for 200ms. Note: Because LabVIEW is inherently multithreaded any other process may continue to function during the execution pause.

Controlling Flow

In LabVIEW,individual code segments can run in both series and parallel. This ability is due to the nature of LabVIEW using multiple threads.

The APT ActiveX interface is a single threaded in process server and as such cannot be accessed through multiple threads. i.e. parallel calls cannot be made. A example of just such a parallel call attempt is shown in Figure 6-22.

Depending on a number of factors, including the number of physical processors, and the way in which the VI was authored, the serial number property may be set before or after the controller communications are started. The order can also vary with each execution. This can be an extremely dangerous situation as code can then appear to be intermittent in correct operation.

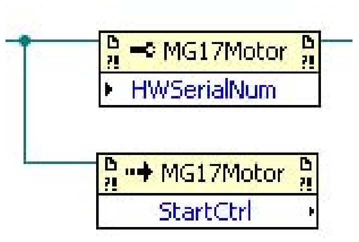


Figure 6-22 – Attempted parallel calls to APT software

Figure 6-23 shows a serial call to the serial number property followed by a call to start controller communications. As all inputs to a node must be satisfied before it can execute and only outputs can be set upon a VI's execution, the start control method execution is held until the serial number property is set first.
By interconnecting APT calls via data flow paths in this way a controlled program flow is achieved.

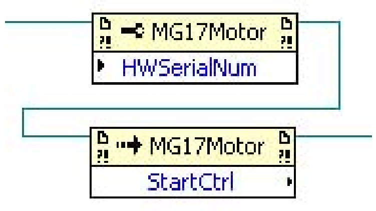


Figure 6-23 – Serial calls to APT

At all times controlled code execution flow should be considered.

In some circumstances it can be beneficial to have parallel segments of code running. For example when performing mathematical calculations which benefit from increased speed.

Error handling

Overview

During authoring of an application, some consideration should be given to handling situations where code fails. This is especially the case when communicating with real pieces of hardware that can be missing, or can exhibit electrical faults, and can also have error responses and conditions of their own.

Although this is not a complete tutorial on error handling, this section covers two aspects specifically associated with ActiveX and APT.

Handling ActiveX errors

ActiveX is a mechanism through which clients and servers may communicate in a windows OS. This communication can at times fail due to invalid file registration, corrupt files and other such problems.
Each call to a server via an ActiveX Inoke or property node has an error cluster input and output.
The purpose of the **error input** is that if there is an error set the node will not be executed.
Any error that occurs during the ActiveX communication will subsequently set the **error output**.

Connecting a series of nodes together with other error handling code ensures that should an error occur a suitable error handling process can be executed.

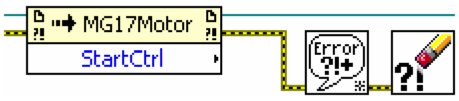


Figure 6-24 – Basic Error Handling

In **Figure 6-24** a very simplified error handler is shown that uses built in LabVIEW VIs. The **General Error Handler** VI and the **Clear Errors** VI are arranged such that if an ActiveX or runtime error occurs a message is displayed reporting the error after which the error cluster is cleared.

Handling APT error return codes

APT methods return an integer which corresponds to an error code. A non zero error code infers that a problem occurred that prevented the method from executing successfully. The specific error code can be used to determine what problem occurred by referring to the APT documentation. A return code of zero infers the method call was successful.

The method return code however needs to be inserted into the standard LabVIEW error cluster in order to be acknowledged by other VI's that have an error input. The basic error handling for the ActiveX example shown in Figure 6-24 as it stands ignores the method return code. So the ActiveX call may be successful but the method may fail, without this condition being captured. However the code shown in Figure 6-25 inserts the error return code into the error cluster.

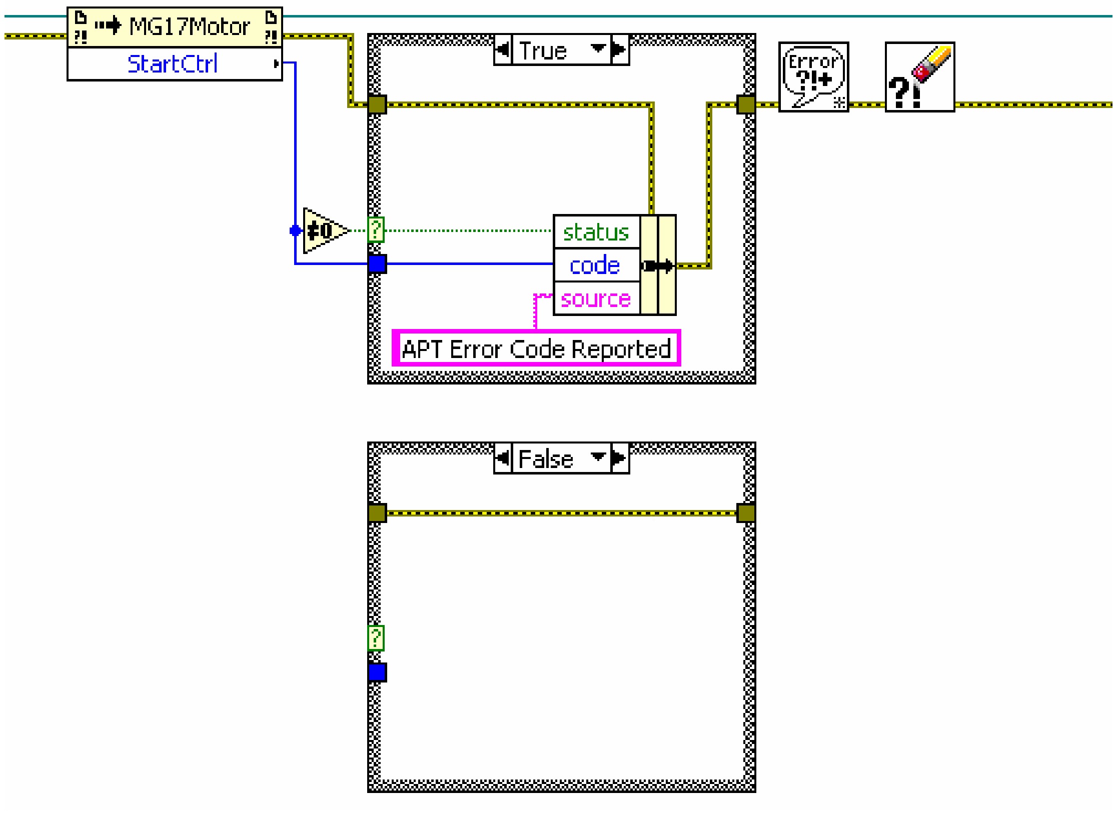


Figure 6-25 – Error handling using the APT return code

In the above error handling scenario, if the return code is zero the result of the **not zero** logic gate will result in a false output. The case structure will then execute the false case contents, which simply passes on the error free data line to following code.

Should the method return an error code indicating that something has gone wrong, the **not zero** logic gates output will be true. The case structure will then execute the true case contents which change the error data cluster contents in the following way;

Error Status is set to true indicating an error.
Error Code is set to the error code returned by the APT method call.
Error source is set to indicate that the error refers to a call to APT.

Tip: Error source information could be elaborated to explain the exact location in code where the error occurred. Also, the error code returned by APT could have an error offset added to the code so not to conflict with other error codes produced by other sections of code within the application.

The code required to incorporate method return codes into the error cluster could be formed as a custom VI.

TIP: It is usually more convenient to form a VI for each method call which incorporates this handling together with default method parameter values.

Complete the following steps to create a return code insertion VI.

1) Create a new VI by selecting from the menu bar **File/New VI**.

2) Right click the Front Panel workspace to display the Controls Palette then select **Modern/Array, Matrix & Cluster**.

3) Select the **Error In 3D.ctl** icon to attach it to the cursor. Then drop the VI onto the front panel.

4) Right click the Front Panel workspace to display the Controls Palette then select **Modern/Array, Matrix & Cluster**.

5) Select the **Error Out 3D.ctl** icon to attach it to the cursor. Then drop the VI onto the front panel.

6) Right click the Front Panel workspace to display the Controls Palette then select **Modern/Numeric**.

7) Select the **Numeric Control** icon to attach it to the cursor. Then drop the VI onto the front panel.

8) Rename the numeric control name by double clicking the default name "**Numeric**" and change to "**Return Code**".

9) Right click the numeric control. From the shortcut menu displayed select **representation/I32** to change the data format of the numeric control to a 32 bit integer.

10) Arrange each of the controls as per Figure 6-26.

11) Select File/Save from the top menu bar, and save the VI in a suitable location. Call the VI "**Return Code Insertion.vi**".

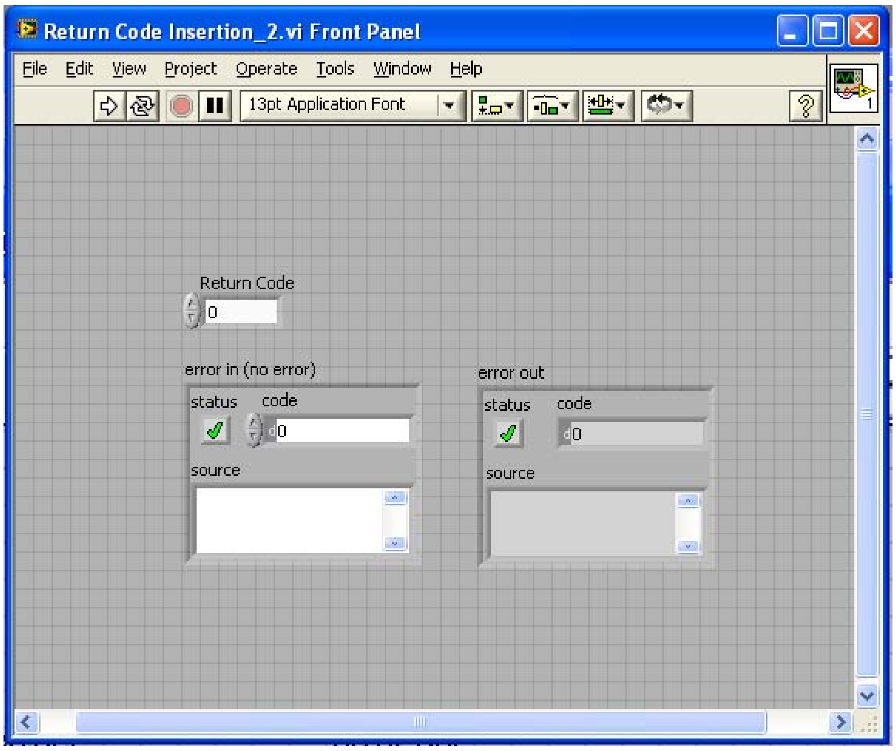


Figure 6-26 – Return code insertion VI front panel

12) Right click on the VI icon at the top right of the Front Panel window. Select '**Show connector**' from the shortcut menu.

13) Right click the connector and select '**Patterns'** from the shortcut menu. Choose the indicated pattern as shown in Figure 6-27.

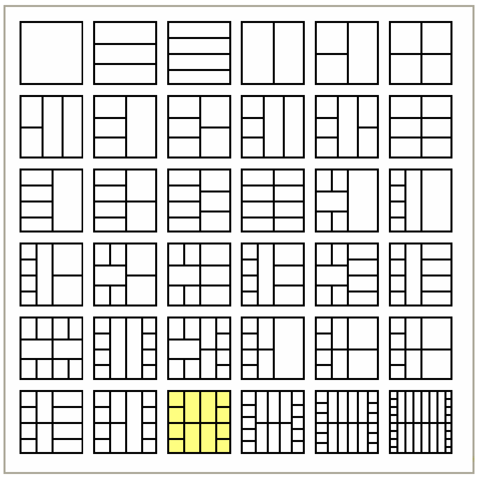


Figure 6-27 – VI Connector Palette

14) Click inside the **top left** square of the connector pattern, then immediately click on the **Return code** control on the front panel. The square on the connector will turn blue.

15) Click inside the **bottom left** square of the connector pattern, then immediately click on the **error in** control on the front panel. The square on the connector will turn a dull green.

16) Click inside the **bottom right** square of the connector pattern, then immediately click on the **error out** control on the front panel. The square on the connector will turn a dull green.

17) The resulting VI connector should appear as shown in Figure 6-28.

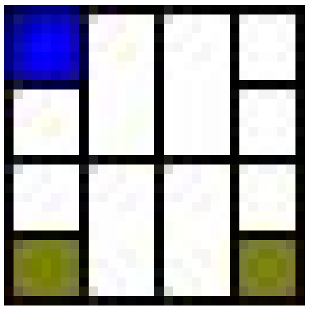


Figure 6-28 – VI Icon Switched to Terminal View

18) Right click the VI wiring connector and select from the shortcut menu **Show Icon**.

19) From the menu bar select **Window / Show Block Diagram**. The block diagram is then shown.

20) Rearrange the three terminals in the block diagram as per Figure 6-29.

21) Right click the white workspace area of the block diagram to display the Functions palette, then select **Programming/Structures**.

22) From the structures palette, select the **Case Structure** icon to attach it to the cursor. Click once in the block diagram to specify the first corner of the case structure and move the mouse to create a rectangular shape. Click once again to specify the opposite corner. Refer to Figure 6-29 for case structure location.

23) Click on the **error in** terminal output connector, move the cursor to draw a straight wire to connect to the left hand edge of the **case structure**. Click on the left hand edge to complete the wire.

24) Click on the **return code** terminal output connector, move the cursor to draw a straight wire to connect to the left hand edge of the **case structure**. Click on the left hand edge to complete the wire.

25) Right click the white area of the block diagram to display the **Functions pale**tte, then select **Programming/Comparison** to display the **Comparison palette**.

26) Select the **Not Equal to 0?** icon to attach it to the cursor. Click in the **block diagram** to drop the VI to the left of the **case structure**. Refer to Figure 6-29 for location.

27) Click on the **Return Code** terminal output connector in the **block diagram** to begin wiring. Click on the input connector of the **Not Equal to 0?** VI to complete the wiring.

28) Click on the **Not Equal to 0?** VI output connector in the **block diagram** to begin wiring. Click the case selector input of the **case structure** to complete the wiring.

29) The VI at this stage should look something like Figure 6-29.

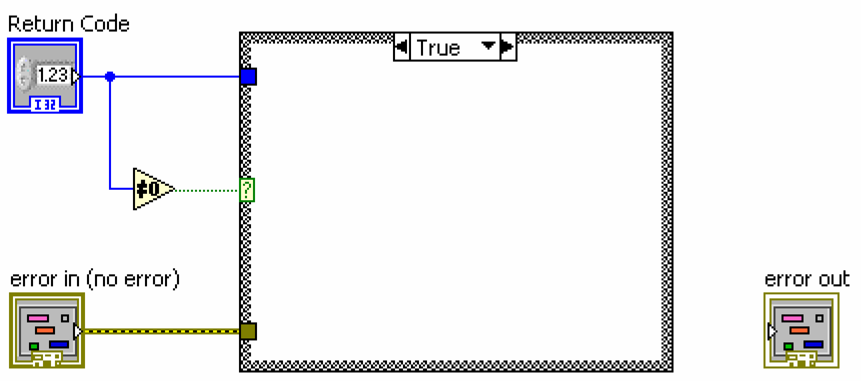


Figure 6-29 – Return Code VI Stage 1 build

30) Right click the white area of the **block diagram** to display the **Functions palette**, then select **Programming/Cluster, Class and Variant**.

31) Select the **Bundle By Name** icon to attach it to the cursor. Click in the **block diagram** to drop the VI inside the **case structure** (TRUE State).

32) Click on the **error in** wire attachment point on the left hand edge of the **case structure** to begin wiring. Click on the input cluster connector on the **Bundle By Name** Function to complete the wiring.

33) Click on the output cluster connector on the **Bundle By Name** Function to begin wiring. Click on the right hand edge of the **case structure** to complete the wiring.

34) Right click on the element name (status) shown on the **Bundle By Name** Function. Select **Add Element**. Repeat this action to add another element.

35) Right click the first element name and select from the shortcut menu **Select Element/Status**.

36) Right click the second element name and select from the shortcut menu **Select Element/Code**.

37) Right click the third element name and select from the shortcut menu **Select Element/Source**.

38) Click on the **case structure** selector on the left hand edge to begin wiring. Click on the **Status** element on the **Bundle By Name** Function to complete the wiring.

39) Click on the **Return Code** connector block on the left hand edge of the case structure to begin wiring. Click on the **Code** element on the **Bundle By Name** Function to complete the wiring.

40) Right click on the **source** element of the **Bundle By Name** Function and select from the shortcut menu **Create/Constant**.

41) Double click on the string constant created and edit to "**APT Error**"

At this stage the VI should look something like Figure 6-30. This completes the wiring for the 'True' part of the code.

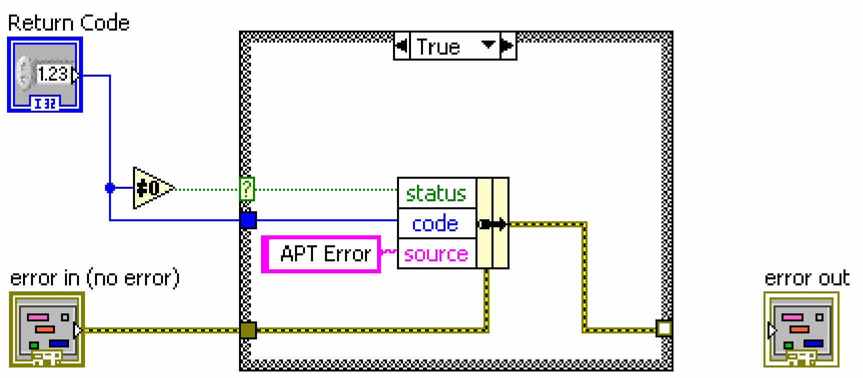


Figure 6-30 – Return Code VI Stage 2 Build

42) Click on the right arrow button on the top edge of the **case structure** to the right of the **True** state. The **False** state should now be shown which is empty as no code has been inserted.

43) Click on the 'error in' connector block on the left hand edge of the **case structure** to begin wiring. Click on the connector block on the right hand edge that was created in the True case window. Clicking on the connector block to compete the wiring will turn the connector from green with a white centre to solid green indicating all cases are wired to this output connector.

44) Click on the error connector block on the right hand edge of the **case structure** to begin wiring. Click on the input connector on the **error out** terminal to complete the wiring.

45) The VI should now look like Figure 6-31.

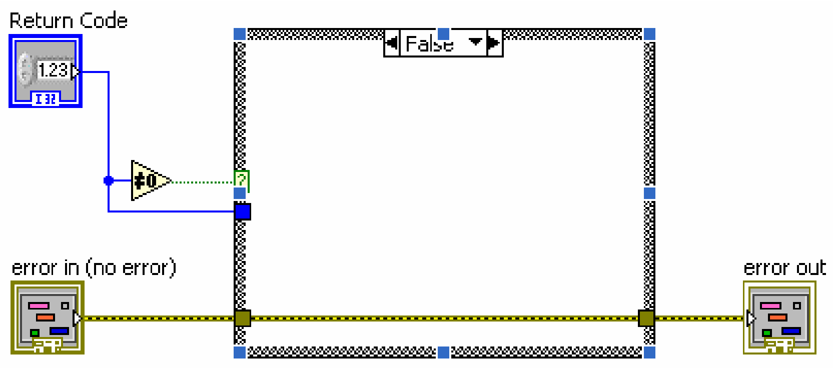


Figure 6-31 – Return Code Insertion Vi Stage 3 Build

46) Save the VI. This completes the authoring of the return code insertion VI.

This VI can be inserted into other higher level VIs to ensure ActiveX method calls to APT inject the APT error return code into the standard LabVIEW error data line.

Adding the APT Error Return Code to the Error Data Line

Using the previously created VI (Return Code Insertion.vi) this can be inserted into the example that has been created so far to add error handling for APT method error codes.

Note. Further sub VIs could be formed to make code easier to follow and more compact however in the example here this will not be done to allow easier following of the example.

Complete the following steps to add basic error handling for APT method errors to the example VI.

1) Return to the Example VI block diagram by selecting from the menu bar **Window/Example.vi Block Diagram**.

2) Right click on the block diagram in some white space and select from the shortcut menu displayed **Select a VI**. Using the file browser locate the previously saved **Return Code Insertion VI**. This attaches the VI to the cursor.

3) Drop the VI between the **StartCtrl** ActiveX method node and the **case structure**, and above the reference line as shown in Figure 6-32.

4) Repeat the insertion of the **Return Code Insertion** VI in the second and third positions as indicated in Figure 6-32.

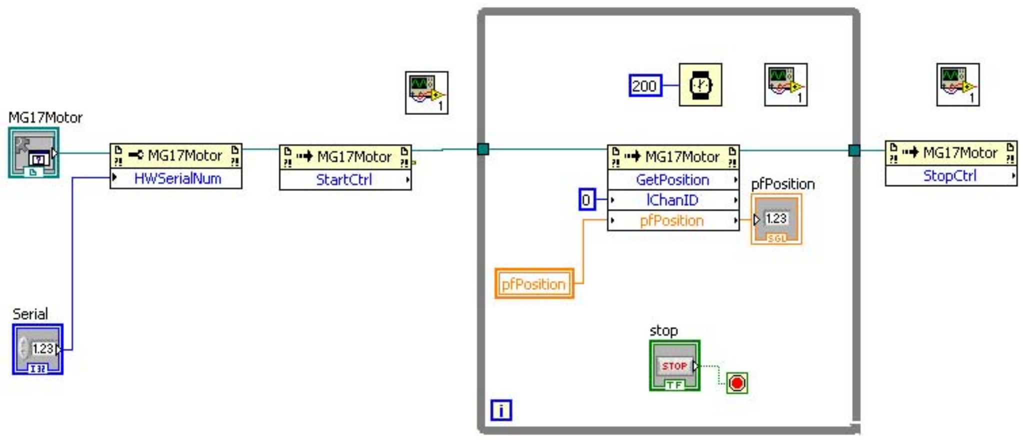


Figure 6-32 – Return Code Insertion VIs added to Example VI

5) Right click the left hand edge of the **while loop** and select from the shortcut menu **Add Shift Register**. This adds a shift register to the **while loop**. The shift register takes data from a previous loop and feeds it back into the next loop.

6) Right click the error input connector on the HWSerialNum ActiveX property node and select from the shortcut menu **Create/Constant**. This will create a default no error state constant to initialise the error cluster line. This isn't strictly necessary, but it does ensure easier to read software.

7) Click the **error output** connector on the **HWSerialNum** ActiveX property node to begin wiring. Click the **error in** connector on the **StartCtrl** ActiveX invoke node to complete the wiring.

8) Click on the **error out** connector on the **StartCtrl** ActiveX invoke node to begin wiring. Click the **error in** connection on the first **Return Code insertion** VI to complete the wiring.

9) Click on the return value output of the **StartCtrl** ActiveX invoke node to begin wiring. Click the **Return Code** input connector of the first **Return Code Insertion** VI to complete the wiring.

10) Click on the **error out** connector of the first **Return Code Insertion** VI to begin wiring. Click the shift register connector block on the left hand edge of the while structure to complete the wiring.

11) Click the shift register connector block on the left hand edge of the **while loop** structure to begin wiring. Click the **error in** of the **GetPosition** ActiveX invoke node to complete the wiring. The diagram should now look like Figure 6-33.

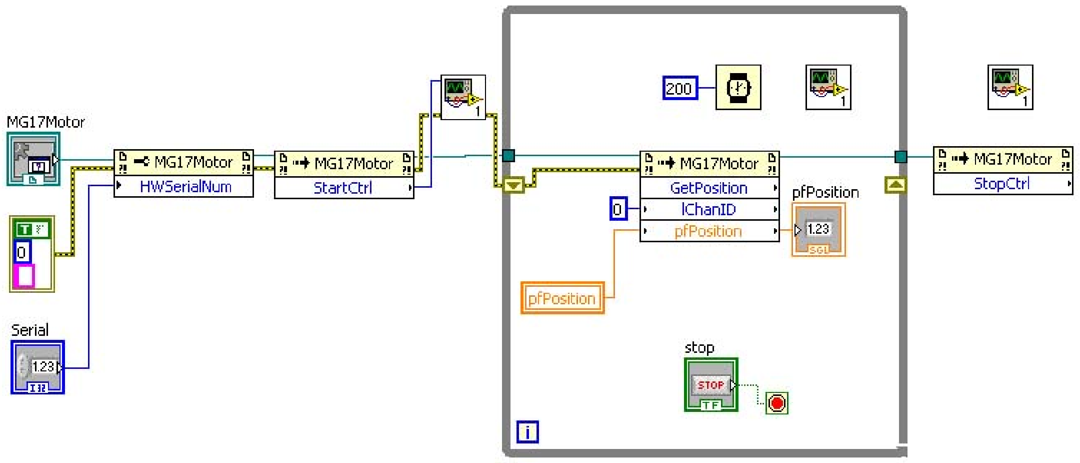


Figure 6-33 – First Return Code Insertion VI Wired Up

12) Click on the **error out** connection of the **GetPosition** ActiveX method node to begin wiring. Click the **error in** connection of the second **Return Code Insertion** VI to complete the wiring.

13) Click on the return value output of the **GetPosition** ActiveX invoke node to begin wiring. Click the **Return Code** input connector of the second **Return Code Insertion** VI to complete the wiring.

14) Click on the **error out** connector of the second **Return Code Insertion** VI to begin wiring. Click the shift register connector block on the right hand edge of the while structure to complete the wiring.

15) Click on the shift register connector block on the right hand edge of the **while loop** to begin wiring. Click the **error in** connector of the **StopCtrl** ActiveX invoke node to complete the wiring – see Figure 6-34.

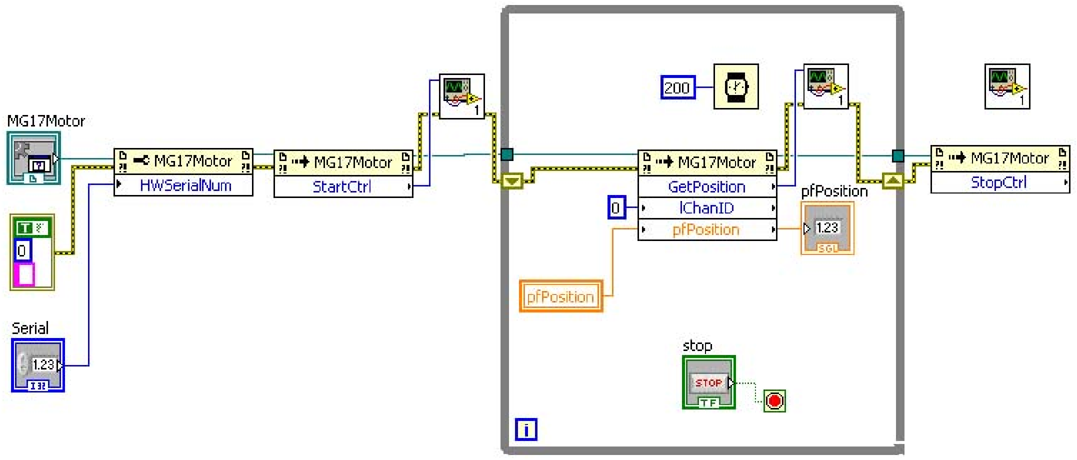


Figure 6-34 – Second Return Code Insertion VI Wired Up

16) Click on the **error out** connector of the **StopCtrl** ActiveX invoke node to begin wiring. Click on the error in connector of the third **Return Code Insertion** VI to complete the wiring.

17) Click on the return value output of the **StopCtrl** ActiveX invoke node to begin wiring. Click the **Return Code** input connector of the third **Return Code Insertion** VI to complete the wiring.

18) Right click the **error out** connector of the third **Return Code Insertion** VI and select from the shortcut menu **Create/ Indicator**. This creates a corresponding front panel indicator that allows any resulting error to be displayed whilst running the VI – see Figure 6-35.

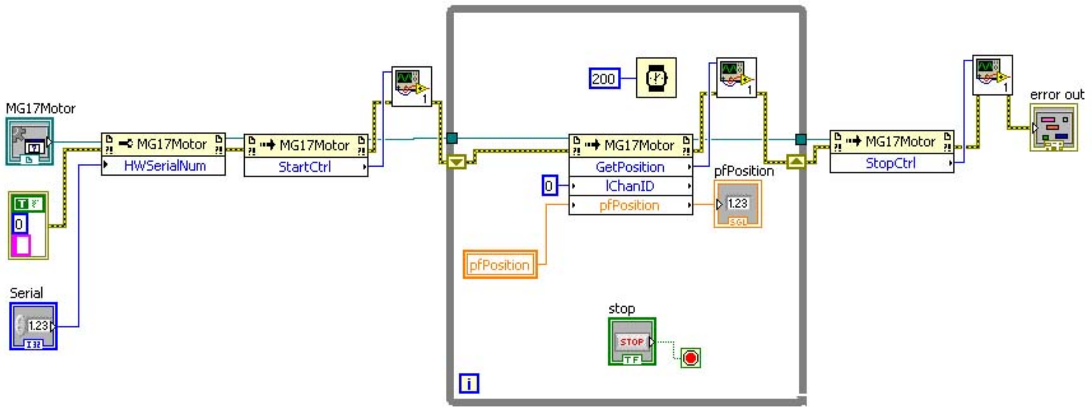


Figure 6-35 – Third Return Code Insertion VI Wired Up

There is one more additional piece of code that needs to be added to enable the while loop to terminate in case of an error. At present if an error occurred whilst in the while loop it would continue until the stop button was pressed.

To add the ability to stop the while loop on demand, or when an error occurs, some **OR** logic needs to be added. To do this the wiring between the **stop** button terminal and the **while loop** stop node needs to be removed and replaced with some Boolean logic.

19) Display the **Cluster, Class & Variant palette** by right clicking the white area of the block diagram and select from the shortcut menu **Programming/Cluster, Class & Variant**.

20) Select the **Unbundle by Name** icon to attach it to the cursor. Click in the block diagram to drop the VI inside the **while loop** structure.

21) Click on the error wire connected to the output connector on the third **Return Code Insertion** VI to begin wiring. Click on the **Unbunble By Name** node input connector on the left hand side to complete the wiring. At this point the **Unbundle by Name** VI will show the first member of the error cluster which is the Boolean error status.

22) Display the **Boolean** palette by right clicking the white workspace area of the block diagram and select from the shortcut menu **Programming/Boolean**.

23) Select the **OR** icon to attach it to the cursor. Click in the **block diagram** to drop the VI inside the **while loop** structure.

24) Click on the wiring between the **stop** button terminal and the while loop stop node, this should select it, and press the **delete key** to remove. Depending upon how the wiring was routed smaller parts of the wiring may remain, and can be selected and removed in the same way. Ensure all parts of the wire are removed.

25) Click on the **status** element of the **Unbundle by Name** node within the **while loop** to begin wiring. Click on the **x** input connector of the **OR** function to complete the wiring.

26) Click on the output connector of the **stop** button terminal to begin wiring. Click on the **y** connection input of the **OR** function to complete the wiring.

27) Click on the output connector of the **OR** function to begin wiring. Click on the while loop **stop** node to complete the wiring.

28) The resulting Example VI should look something like that in Figure 6-36.

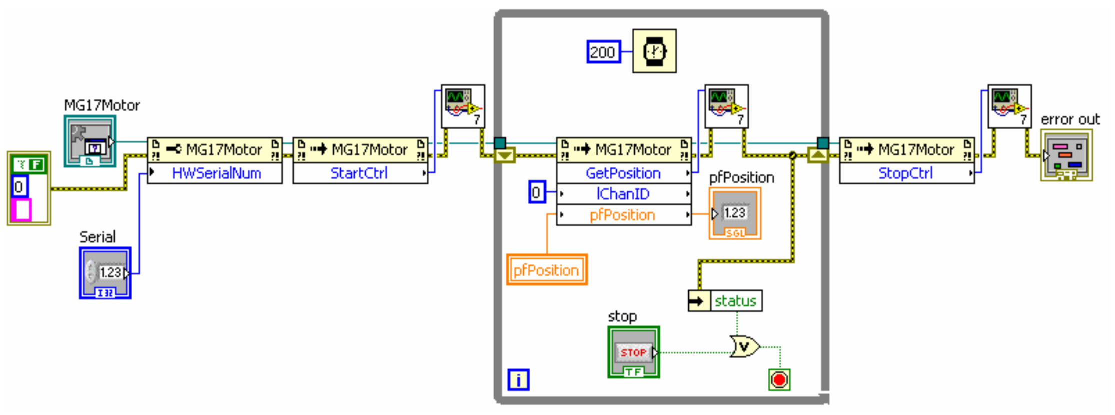


Figure 6-36 – Example VI with basic error handling

Running a VI with APT

The **Example** VI can be run from either the block diagram or the front panel. It is preferential in most cases to run from the front panel as all the visible controls are located within this window.

For this particular explanation of how to run the VI it is assumed that we are communicating with a simulated DC motor controller (TDC001) with a serial number of 83123456. To configure the APT software for a particular hardware configuration, refer to the APT software help files and documentation.

Complete the following steps to start, operate and stop the VI.

1) If not already viewing the front panel of the Example VI select from the menu bar **Window/Example.vi Front Panel** to switch the view.

Note: If using real hardware, the 8 digit serial number entered at item (3) will depend upon the individual controller you are using.
For the purposes of this tutorial, a simulated configuration will be used to run the VI, based on the TDC001 unit. If using real hardware, ignore item (2).

2) Run the APTConfig utility and set up a simulated configuration using a TDC001 unit. Refer to the help file (Start/Programs/Thorlabs/APT/Help/APTConfig Help) for instructions on how to set up a simulated configuration.

3) If required - Click on the serial number control window to set the serial number of the controller. Edit the serial number to 83123456.

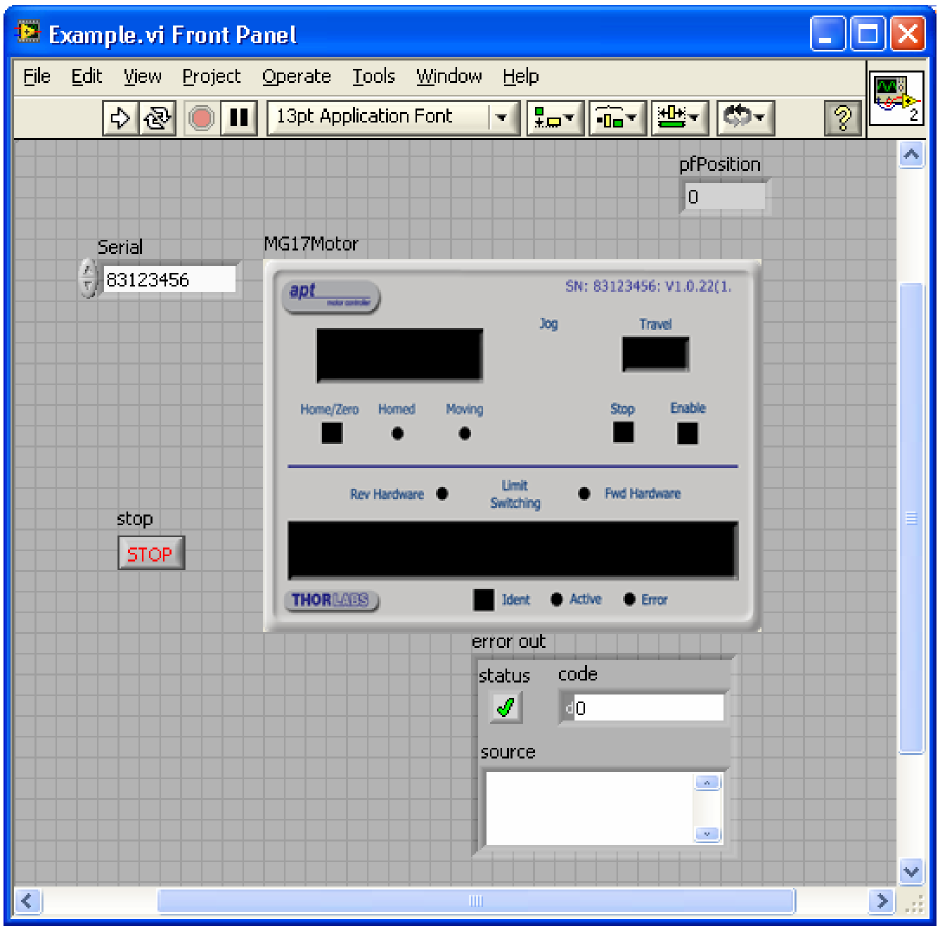
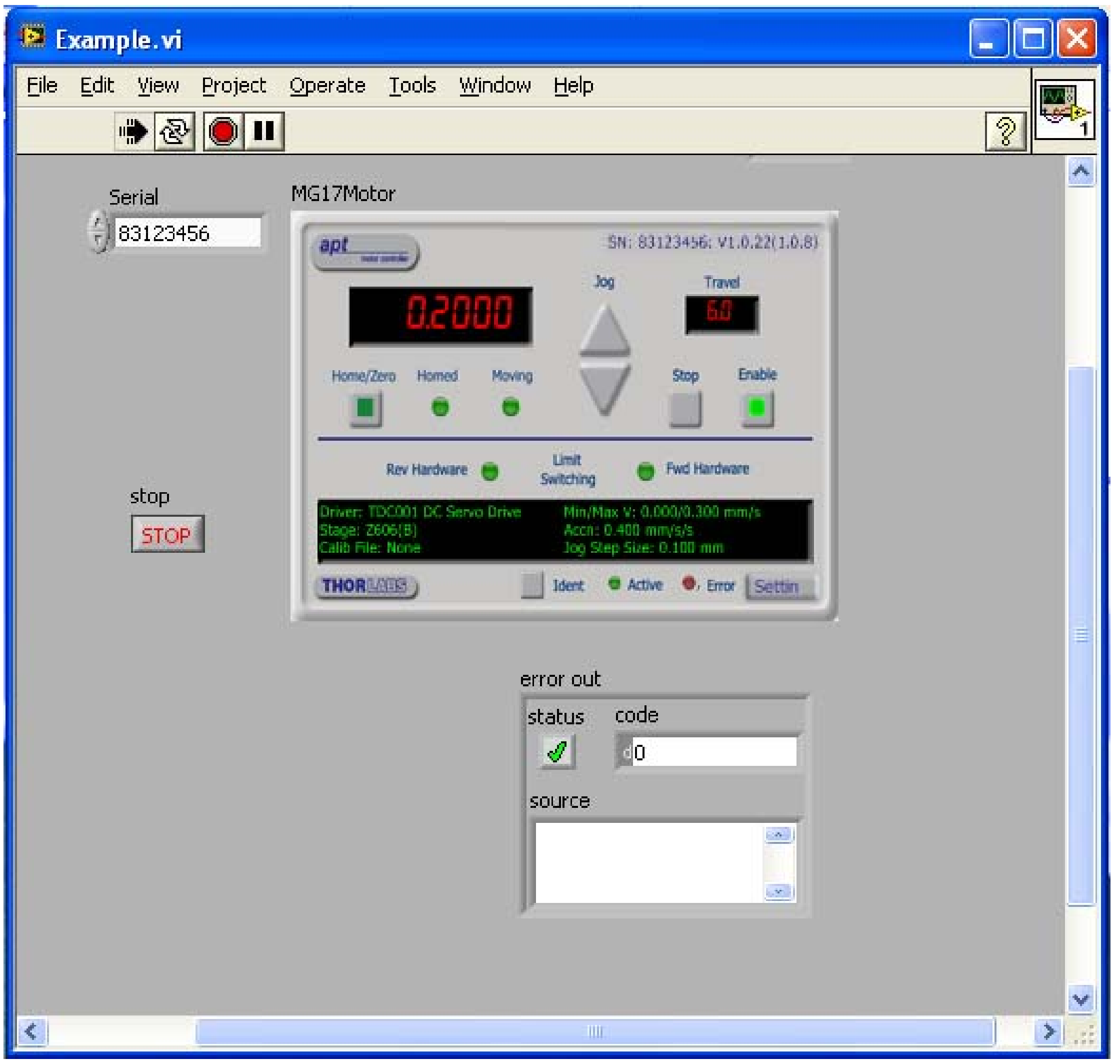


Figure 6-37 – Example VI with serial number entered

4) Click the ‘**Run’** button in the **Front Panel** menu bar. DO NOT click the ‘Run Continuously’ button.

5) The VI will start and there will be a short delay of several seconds whilst communication with the APT controller is established.

6) Once communication is established the virtual control panel of the APT controller will become activated and will display live data. e.g. current position.



```
Driver: TDC001 DC Servo Drive        Min/Max V: 0.000/0.300 mm/s
Stage: Z606(B)                       Accn: 0.400 mm/s/s
Calib File: None                     Jog Step Size: 0.100 mm
```

Figure 6-38 – Example VI with live data

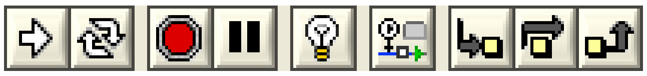
Pressing the jog buttons of the APT virtual control panel - notice that the position indicator on the front panel above the virtual panel also is updated. This indicator is populated from information gathered from the APT system via ActiveX as per the block diagram.

7) Press the stop button control on the front panel to terminate the VI. Do not stop the VI using the stop button on the menu tool bar.

Pressing the stop button terminates the block diagram while loop and executes the ActiveX call to stop communication. Finally the VI is stopped and the run button is released.

Within the LabVIEW development environment it is possible to stop execution by pressing the stop button on the menu toll bar. It should be noted that in stopping the VI in this manner does not stop communication to the APT controls and any such disconnection of the equipment will result in an error dialogue.

Debugging a VI



- Run Button.
- Run Continuous Button.
- Stop Button.
- Highlight Execution Button.
- Retain Wire Values Button.
- Step Into Button.
- Step Over Button.
- Step Out Button.

Figure 6-39 – Execution Controls on the menu tool bar

Running a VI with Highlighted Execution

It is possible to view an animation of the execution of the block diagram by clicking the **Highlight Execution** menu tool button, shown in Figure 6-39. Highlighted execution shows the movement of data within the block diagram from one node to another using popup bubbles that move along the wires. Use execution highlighting in conjunction with other techniques to see how data moves from one node to the next.

Note Highlighted execution greatly reduces the speed at which a VI runs.
If the **error out** cluster reports an error, the error value appears next to **error out**. If no error occurs, **OK** appears next to **error out**.

To examine the example VI in highlighted execution follow the steps below.

1) Arrange both the **block diagram** and the **front panel** such that they are both visible.

Note: Select from the menu bar **Windows/Show Front Panel** if the front panel is not visible. Similarly if the block diagram is not visible select **Windows/Show Block Diagram**.

2) Click the **highlighted execution** menu tool bar button in the **block diagram** so that the light bulb icon is illuminated.

3) Press the **Run** button to begin execution.

During execution observe the popup bubbles that display data as each node is entered and exited. Also note the slow execution.

4) Press the **stop** button control on the front panel to stop execution. Do not press the stop button on the menu tool bar.

5) The VI will terminate its **while loop**, stop communications and stop execution.

6) Click the highlighted execution menu tool bar button in the block diagram so that the light bulb icon is extinguished. This restores the VI to normal execution.

Observing the animation of the executing VI can reveal a number of unexpected problems and is one of the first tools to deploy in the event of an unexpected result or error.

Stepping Through a VI

Single-step through a VI to watch and evaluate each step within the VI on the block diagram The single-stepping buttons, shown in Figure 6-39 affect execution only in a VI or subVI in single-step mode.

Enter single-step mode by clicking the **Step Over** or **Step** Into menu tool bar button on the block diagram toolbar. Subsequent presses of the Step Into, Step Over and Step Out will have the following actions.

Step Into - Executes the next action within the block diagram or enters a sub-VI.

Step Over - Executes the next action within the block diagram or complete sub VI.

Step Out - Executes the remainder of the current VI.

To observe the stepping through the example VI complete the following steps.

1) Arrange both the **block diagram** and the **front panel** such that they are both visible.

Note: Select from the menu bar **Windows/Show Front Panel** if the front panel is not visible. Similarly if the block diagram is not visible select **Windows/Show Block Diagram**.

2) Click the **Step Into** menu toolbar button in the **block diagram** so to begin execution. Notice that the pause button is illuminated and that the first action is flashing. This indicates that the VI is paused and that the first action as executed.

3) Press the **Step Into** button twice more.

4) The virtual control panel should now be active as both the serial number property node and start control node have been executed. The VI is again paused and the next action is flashing.

5) Press the **Step out** menu toolbar button to continue to execute to the end of the example VI.

6) Press the **stop** button control on the **front panel** to terminate the VI.

7) Notice that the VI doesn’t terminate completely as the stepping mode has paused the VI once again before exiting.

8) Press the **Step out** button to complete the VI.

Note: Pressing the **Step Over** button allows stepping execution to skip a sub VI should the user be confident that a problem doesn't occur within it.

Probe Tools

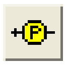


Figure 6-40 – Probe Tool Icon

The Probe tool can be used to check intermediate data on a wire as a VI runs. Use the Probe tool if you have a complicated VI with a number of operations, any one of which might return incorrect data. The probe tool is available from the Tools palette (View/Tools Palette).

If data is available, the probe will immediately update during execution allowing the user to inspect data at specific points within the block diagram.

Types of Probes

You can check intermediate data on a wire when a VI runs by using the Generic probe, by using an indicator from the **Controls** palette to view the data, by using a supplied probe, by using a customized probe

In this document we will only discuss the generic probe. See LabVIEW documentation for further details of other probe types.

Generic probe

Use the generic probe to view the data that passes through a wire. While the VI is paused or stopped right click a wire and select **Probe** from the displayed shortcut menu.
The generic probe will display the data appropriately.

To examine the use of the probe tool complete the following steps.

1) Open and arrange both the **block diagram** and the **front panel** such that they are both visible.

Note: Select from the menu bar **Windows/Show Front Panel** if the front panel is not visible. Similarly if the block diagram is not visible select **Windows/Show Block Diagram**.

2) Within the block diagram right click the wire connecting the position output of the **GetPosition** ActiveX invoke node (pfPosition output) to the pfPosition indicator terminal. From the shortcut menu displayed select **Probe**.
3) A probe is automatically created with the correct data type, and in this instance shows a numeric indicator control. See Figure 6-41.

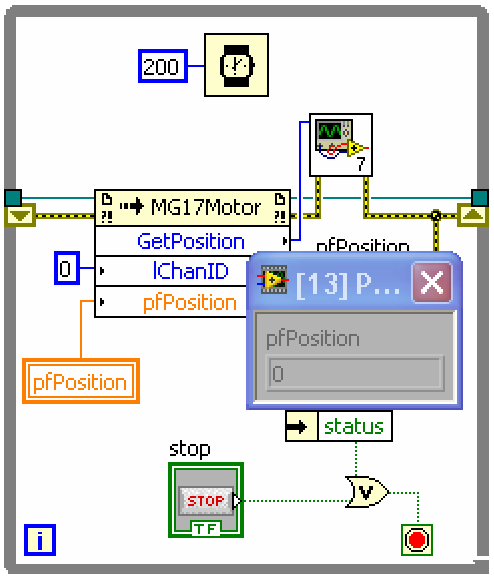


Figure 6-41 – Generic Probe Tool In Use

4) Press the Run button to start the VI.

5) Notice that the **probe tool** text and data now changes colour from a dull grey to black. This indicates that the data displayed has been updated.

6) Press the jog up and down buttons on the APT virtual control panel to move the motor and notice that the probe displays the same data. As the **while loop** is repeatedly executed the probe samples the data on the wire it is associated with on each execution.

7) Press the **stop** button control on the front panel to stop the VI.

8) Press the red close cross on the probe to remove it.

Breakpoints

Figure 6-42 – Breakpoint Tool Icon

The Breakpoint tool, shown in Figure 6-42, can be used to place breakpoints on a VI, nodes, and wires within the block diagram and pause execution at that location. When you set a breakpoint on a wire, execution pauses after data passes through the wire. Place a breakpoint on the block diagram to pause execution after all elements in the block diagram have executed. The Breakpoint tool is available from the Tools palette (View/Tools Palette).

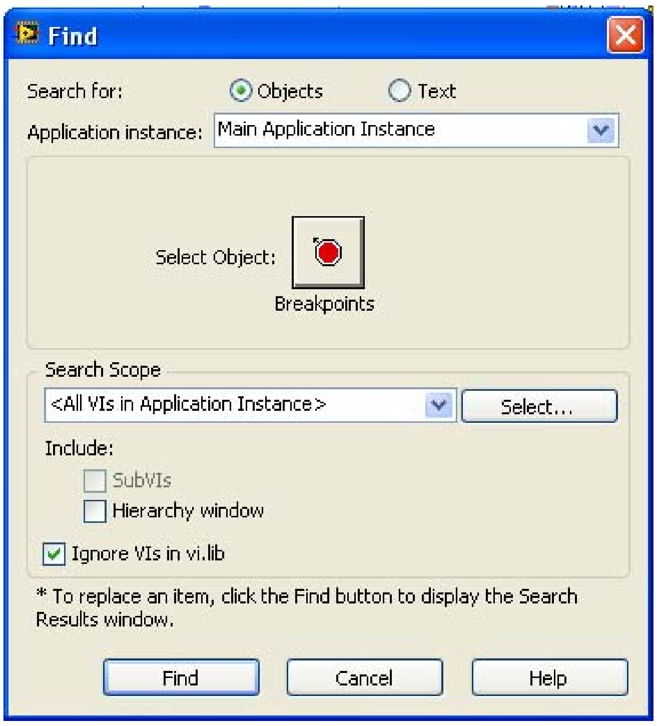
When a VI pauses at a breakpoint, LabVIEW brings the block diagram to the front and highlights the node or wire that contains the breakpoint.

When you reach a breakpoint during execution, the VI pauses and the **Pause** button on the menu toolbar appears red. You can then take the following actions:

• Single-step through execution using the single-stepping buttons.

• Probe wires to check intermediate values.
• Change values of front panel controls.
• Click the **Pause** button to continue running to the next breakpoint or until the VI finishes.

LabVIEW saves breakpoints within a VI. You can view all breakpoints by selecting **Operate/Breakpoints** from the main menu bar and then pressing the **find** button in the dialogue window that opens.


Find

Search for: [✓ Objects] Text

Application instance: Main Application Instance

Select Object: Breakpoints

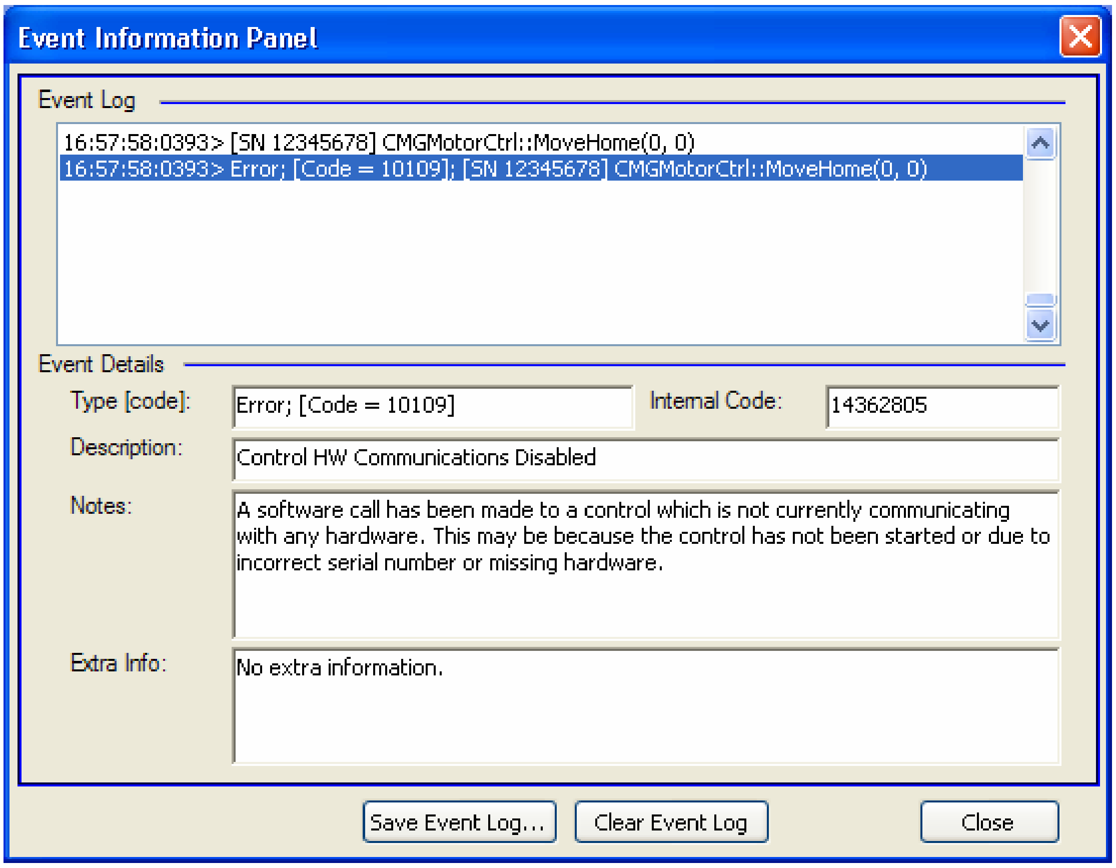
Search Scope
<All VIs in Application Instance>

Include:
☐ SubVIs
☐ Hierarchy window
☑ Ignore VIs in vi.lib

* To replace an item, click the Find button to display the Search Results window.

Figure 6-43 – Finding Breakpoints

APT Event Dialogue



```
16:57:58:0393> [SN 12345678] CMGMotorCtrl::MoveHome(0, 0)
16:57:58:0393> Error; [Code = 10109]; [SN 12345678] CMGMotorCtrl::MoveHome(0, 0)
```

Type [code]: Error; [Code = 10109]

Internal Code: 14362805

Description: Control HW Communications Disabled

Notes: A software call has been made to a control which is not currently communicating with any hardware. This may be because the control has not been started or due to incorrect serial number or missing hardware.

Extra Info: No extra information.

Figure 6-44 – Example Event Dialogue Window

The **APT Event Information Panel** is a window that allows the developer to understand why a particular event occurred. An event could be a warning or an error.

Method calls made into the APT software and operation of the APT system as a whole are continuously monitored for problems. Should a warning condition or error occur, a dialogue window is displayed such as that in Figure 6-44.

The window shows a time stamped series of events up to and including the warning or error condition.

The event is described for the developer as well as giving notes and extra information as to the possible cause. Should the cause of the event not be understood clearly the event log can be saved and passed to technical support for analysis and further assistance.

See the APT programming help file for further information on event reporting configuration, and for disabling for deployed applications.

Chapter 7 - Event Driven Programming

What are Events?

An event is an asynchronous notification that something has occurred. Events can originate from hardware, the user interface or other parts of the program. User interface events include human interface devices such as mouse clicks and key presses. Hardware events occur when some situation has arisen or when an error condition occurs.
Other types of events can be generated programmatically and used to communicate with different parts of the program.

In an event-driven program, events that occur in the system directly influence the execution flow. In contrast, a procedural program executes in a pre-determined, sequential order. Event-driven programs usually include a loop that waits for an event to occur, executes code to respond to the event, and continues to wait for the next event.

The order in which an event-driven program executes depends on which events occur and on the order in which they occur. Some sections of the program might execute frequently because the events they handle occur frequently, and other sections of the program might not execute at all because the events never occur.

Why Use Events?

Events allow you to execute a specific event-handling case each time a user performs a specific action or a piece of hardware reports a situation.
Without events, the block diagram must poll the state of front panel controls in a loop, checking to see if any change has occurred. Polling the front panel requires a significant amount of CPU time and can fail to detect changes if they occur too quickly.

By using events to respond to specific actions, you eliminate the need to poll the front panel to determine which actions the user performed or hardware has reported. Instead, LabVIEW actively notifies the block diagram each time an interaction you specified occurs. Using events reduces the CPU requirements of the program, simplifies the block diagram code, and guarantees that the block diagram can respond to all actions the user makes or hardware reports.

Registering and Handling an APT ActiveX Event

When handling an ActiveX event, you must register for the event and then create a callback VI to handle that event.

Complete the following steps to register and handle an ActiveX event in LabVIEW.

1) If not already, open the **Example** VI.

2) View the block diagram of the example by selecting from the menu bar **Window / Show Block Diagram**.

3) Display the **ActiveX** palette by right clicking the block diagram workspace to display the functions palette and then navigating to **Connectivity/ActiveX**.

4) Select the **Register Event CallBack** icon to attach it to the cursor. Then drop the VI onto the **block diagram**.

5) Click the wire connected to the **reference out** node of the **StartCtrl** invoke node to begin wiring. Click the **event source ref** input of the **Register Event Callback** function to complete the wiring.

6) Click the arrow in the 'Event' panel, and select **MoveComplete** from the drop down menu.

7) Right click the **VI Ref** input node and select **Create Callback VI**. This automatically creates a VI with the correct event information.

8) Save the new VI as "**MoveComplete Event Callback.vi**". Select **File/Save As…** from the menu bar, select a suitable location to save the file and enter the VI name. Press Save.

9) The Example VI should now look like Figure 7-1.

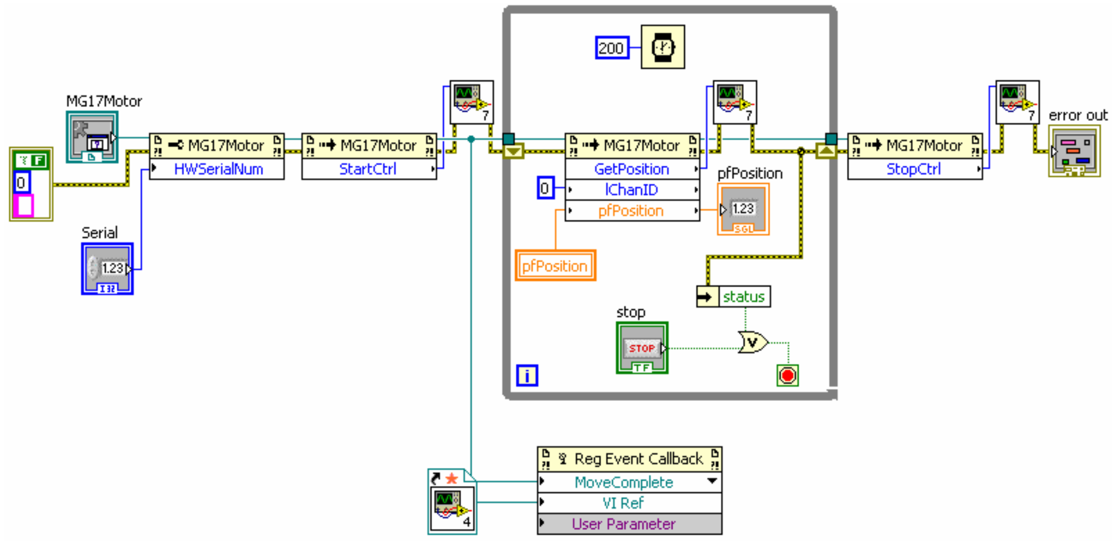


Figure 7-1 – Example VI with Event Callback

Currently there is nothing in the new **MoveComplete Event Callback** VI that will perform any function. Figure 7-2 shows the block diagram contents that have automatically been generated. To see the event call back operating it is necessary to add some code into the VI.

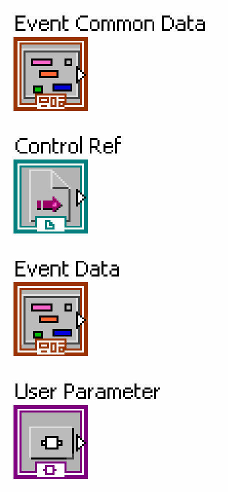


Figure 7-2 – MoveComplete Event Callback VI Block Diagram

10) Double click the call back VI in the Example VI block diagram.

11) From the menu bar of the **MoveComplete Event Callback** VI select **Window / Show Block Diagram**.

12) Right click the output connector of the **Event Data** cluster and select from the shortcut menu **Cluster, Class & Variant/Unbundle By Name**. Then drop the function onto the **block diagram**.

13) Click the output connector of the **Event Data** cluster to begin wiring. Click the input connector of the **Unbundle By Name** function to complete the wiring. The **Unbundle By Name** function will automatically select the first element in the cluster. As this particular event has only one event parameter this will suffice.

14) Right click the block diagram white space to display the Functions palette and navigating to **Programming/String** in the shortcut menu. Select the **Format Into String** icon to attach it to the cursor. Then drop the function onto the block diagram.

15) Click on the output connector of the **Unbundle By Name** (lChanID) function to begin wiring. Click on **Input 1** of the **Format Into String** function to complete the wiring.

16) Right click the **Initial String** connector on the **Format Into String** function and from the shortcut menu select **Create / Constant**.

17) Edit the string constant created, by double clicking the contents, and enter the text "**Move Complete on Channel** ", including the space after the word 'channel'.

18) Right click the block diagram white space to show the **Functions palette**, and navigating to **Programming/Dialogue & User Interface** in the shortcut menu. Select the **One Button Dialogue** icon to attach it to the cursor. Then drop the function onto the block diagram.

19) Click the **resulting string** output connector of the **Format Into String** function to begin wiring. Click the **message** input connector of the **One Button Dialogue** function to complete the wiring.

20) The VI block diagram should now look something like Figure 7-3.

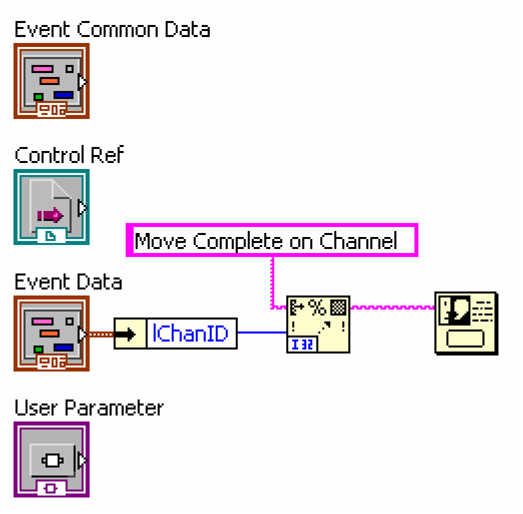


Figure 7-3 - MoveComplete Event Callback VI Block Diagram

When the **Example** VI is run, any time a movement of the controller occurs and stops the move complete event will be fired which in turn will make a call back to the VI registered for the event.

21) Return to the **Example** VI front panel by selecting from the menu bar **Window / Example.vi Front Panel**.

22) Run the VI by pressing the **run** button on the menu toolbar.

23) Once the APT virtual control panel has become active and is displaying the current position press the up and down arrows on the panel to move the motor attached. Notice that every time there is a jog movement a message prompt is created which needs to be acknowledged by pressing Ok.

The message prompt is displayed due to the event making a call to the code contents of the **MoveComplete Event Callback** VI. Although this in itself is not entirely useful this idea can be expanded upon to handle multiple events and to execute more interesting blocks of code.

Glossary

Block Diagram	*Pictorial description or representation of a program. The block diagram consists of executable icons called nodes and wires that carry data between the nodes. The block diagram is the source code for the VI.*
Boolean controls and indicators	*Front panel objects to manipulate and display Boolean (TRUE or FALSE) data.*
Breakpoint	*Pause in execution point within a VI used for debugging.*
Breakpoint Tool	*Tool to set a breakpoint on a VI, node, or wire.*
Cluster	*A set of ordered, none indexed data elements of any data type, including numeric, Boolean, string, array, or another cluster. The elements must be all controls or all indicators.*
Connector	*Part of the VI or function node that contains input and output terminals. Data passes to and from the node through a connector.*
Control	*Front panel object for entering data into a VI interactively, such as a knob or push button.*
Data Type	*Format of information. In LabVIEW, acceptable data types for most VIs and functions are numeric, array, string, Boolean, refnum, enumeration, and cluster.*
Error In	*Error structure that enters a VI.*
Error Message	*Indication of a software or hardware malfunction or of an unacceptable data.*
Error Out	*The error structure that leaves a VI.*
Error Structure	*Consists of a Boolean status indicator, a numeric error code indicator, and a string source indicator.*
Execution Highlighting	*Debugging technique that animates VI execution to illustrate the data flow in a VI.*
For Loop	*Iterative loop structure that executes a block of code a set number of times.* *Equivalent to text-based code: For i = 0 to n – 1, do...*
Front Panel	*Interactive user interface of a VI.*
Functions Palette	*Palette that contains VIs, functions, block diagram structures, and constants.*
Icon	*Graphical representation of a node in a block diagram.*
Menu Bar	*Horizontal bar that lists the names of the main menus of an application. The menu bar appears below the title bar of a window. Most applications have a menu bar that is distinct for that application.*
Multithreaded Application	*An application that runs several different threads of execution independently. On a multiple processor computer, the different threads might be running on different processors simultaneously.*

Node	*Program execution element. Nodes are analogous to statements, operators, functions, and subroutines in text-based programming languages.* *On a block diagram, nodes include functions, structures, and subVIs.*
Numeric controls and indicators	*Front panel objects to manipulate and display numeric data.*
Object	*Generic term for any item on the front panel or block diagram, including controls, indicators, nodes, and wires.*
OS	*Operating System*
Palette	*Display of icons that represent possible options.*
Probe	*Debugging feature for checking intermediate values in a VI.*
Probe Tool	*Tool to create probes on wires.*
Shift Register	*Optional mechanism in loop structures to pass the value of a variable from one iteration of a loop to a subsequent iteration. Shift registers are similar to static variables in text-based programming languages.*
Shortcut Menu	*Menu accessed by right-clicking an object. Menu items pertain to that object specifically.*
String	*Representation of a value as text.*
String controls and indicators	*Front panel objects to manipulate and display text.*
Structure	*Graphical representation of loops and sequences*
SubVI	*VI used in a block diagram of another VI. Comparable to a subroutine in text based programming applications.*
Terminal	*An object in the block diagram that passes data to and from the front panel*
Tools Palette	*Palette that contains tools you can use to edit and debug front panel and block diagram objects.*
Wire	*Data path between nodes.*
Wire Bend	*Point where two wire segments join.*
Wire Branch	*Section of wire that contains all the wire segments from junction to junction, terminal to junction, or terminal to terminal if there are no junctions between.*
Wire Junction	*Point where three or more wire segments join.*
Wire Segments	*Single horizontal or vertical piece of wire.*
Wire Stubs	*Truncated wires that appear next to unwired terminals when you move the Wiring tool over a VI or function node.*
Wiring Tool	*Tool to define data paths between terminals.*

Short Cut Quick Reference

Objects and Movement

Shift-click	Selects multiple objects; adds object to current selection.
↑↓→← (arrow keys)	Moves selected objects one pixel at a time.
Shift-↑↓→←	Moves selected objects several pixels at a time.
Shift-click (drag)	Moves selected objects in one axis.
Ctrl-click (drag)	Duplicates selected object.
Ctrl-Shift-click (drag)	Duplicates selected object and moves it in one axis.
Shift-resize	Resizes selected object while maintaining aspect ratio.
Ctrl-resize	Resizes selected object while maintaining center point.
Ctrl-Shift-resize	Resizes selected object while maintaining center point and aspect ratio.
Ctrl-drag a rectangle	Inserts more working space on the front panel or block diagram.
Ctrl-A	Selects all items on the front panel or block diagram.
Ctrl-Shift-A	Performs last alignment operation on objects.
Ctrl-D	Performs last distribution operation on objects.
Double-click open space	Places a free label on the front panel or block diagram if automatic tool selection is enabled.
Ctrl-mouse wheel	Scrolls through sub-diagrams of a Case, Event, or Stacked Sequence structure.
Spacebar (drag)	Disables preset alignment positions when moving labels or captions.
Ctrl-U	Reroutes all wires and rearranges block diagram objects automatically.

Debugging

Ctrl ↓	Steps into node.
Ctrl →	Steps over node.
Ctrl ↓	Steps out of node.

Basic Editing

Ctrl-Z	Undoes last action.
Ctrl-Shift-Z	Redoes last action.
Ctrl-X	Cuts an object.
Ctrl-C	Copies an object.
Ctrl-V	Pastes last cut or copied object.

Navigating the LabVIEW Enviroment

Ctrl-E	Displays block diagram or front panel.
Ctrl-#	Enables or disables grid alignment.
Ctrl-/	Maximizes and restores window.
Ctrl-T	Tiles front panel and block diagram windows.
Ctrl-F	Finds objects or text.
Ctrl-G	Searches VIs for next instance of object or text.
Ctrl-Shift-G	Searches VIs for previous instance of object or text.
Ctrl-Shift-F	Displays the **Search Results** window.
Ctrl-Tab	Cycles through LabVIEW windows.
Ctrl-Shift-Tab	Cycles the opposite direction through LabVIEW windows.
Ctrl-Shift-N	Displays the **Navigation** window.
Ctrl-I	Displays the **VI Properties** dialog box.
Ctrl-L	Displays the **Error list** window.
Ctrl-Y	Displays the **History** window.
Ctrl-Shift-W	Displays the **All Windows dialogue box**.
Ctrl-Space	Displays the **Quick Drop** dialogue box.

Navigating the VI Hierarchy Window

Ctrl-D	Redraws the window.
Ctrl-A	Shows all VIs in the window.
Ctrl-click VI	Displays the subVIs and other nodes that make up the VI you select in the window.
Enter †	Finds the next node that matches the search string.
Shift-Enter †	Finds the previous node that matches the search string.

† After initiating a search by typing in the VI Hierarchy window

File Operations

Ctrl-N	Creates a new VI.
Ctrl-O	Opens an existing VI.
Ctrl-W	Closes the VI.
Ctrl-S	Saves the VI.
Ctrl-Shift-S	Saves all open files.
Ctrl-P	Prints the window.
Ctrl-Q	Quits LabVIEW.

Help

Ctrl-H	Displays the **Context Help** window.
Ctrl-Shift-L	Locks the **Context Help** window.
Ctrl-? or F1	Displays the LabVIEW Help.

Tools and Palettes

Ctrl	Switches to next most useful tool.
Shift	Switches to Positioning tool.
Ctrl-Shift over open space	Switches to Scrolling tool.
Spacebar †	Toggles between two most common tools.
Shift-Tab †	Enables automatic tool selection.
Tab †	Cycles through four most common tools if you disabled automatic tool selection by clicking the **Automatic Tool Selection** button. Otherwise, enables automatic tool selection.
↑↓→←	Navigates temporary **Controls** and **Functions** palettes.
Enter	Navigates into a temporary palette.
Esc	Navigates out of a temporary palette.
Shift-right-click	Displays a temporary version of the **Tools** palette at the location of the cursor.
† If automatic tool selection is disabled	

SubVISs

Double-click subVI	Displays subVI front panel.
Ctrl-double-click subVI	Displays subVI block diagram and front panel.
Drag VI icon to block diagram	Places that VI as a subVI on the block diagram.
Shift-drag VI icon to block diagram	Places that VI as a subVI on the block diagram with constants wired for controls that have non-default values.
Ctrl-right-click block diagram and select VI from palette	Opens front panel of that VI.

Execution

Ctrl-R	Runs the VI.
Ctrl-. †	Stops the VI.
Ctrl-M	Changes to run or edit mode.
Ctrl-Run button	Recompiles the current VI.
Ctrl-Shift-Run button	Recompiles all VIs in memory.
Ctrl-↓	Moves key focus inside an array or cluster.
Ctrl ↑	Moves key focus outside an array or cluster.
Tab †	Navigates the controls or indicators according to tabbing order.
Shift-Tab †	Navigates backward through the controls or indicators.
† While the VI is running	

Text	
Double-click	Selects a single word in a string.
Triple-click	Selects an entire string.
Ctrl-→	Moves forward in string by one word.
Ctrl-←	Moves backward in string by one word.
Home	Moves to beginning of current line in string.
End	Moves to end of current line in string.
Ctrl-Home	Moves to beginning of entire string.
Ctrl-End	Moves to end of entire string.
Shift-Enter	Adds new items when entering items in enumerated type controls and constants, ring controls and constants, or Case structures.
Esc	Cancels current edit in a string.
Ctrl-Enter	Ends text entry.
Ctrl-=	Increases the current font size.
Ctrl--	Decreases the current font size.
Ctrl-0	Displays the **Font** dialog box.
Ctrl-1 †	Changes to the Application font.
Ctrl-2 †	Changes to the System font.
Ctrl-3 †	Changes to the Dialog font.
Ctrl-4 †	Changes to the current font.
† In the Font dialog box	

Wiring	
Ctrl-B	Removes all broken wires.
Esc, right-click, or click terminal	Whilst wiring cancels a wire you started.
Single-click wire	Selects one segment.
Double-click wire	Selects a branch.
Triple-click wire	Selects entire wire.
A	While wiring, disables automatic wire routing temporarily.
Double-click	While wiring, tacks down wire without connecting it.
Spacebar	While wiring, switches the direction of a wire between horizontal and vertical.
spacebar	Whilst moving objects, toggles automatic wiring.
Ctrl-click input on function with two inputs	Switches the two input wires.
Shift-click	While wiring, undoes last point where you set a wire.