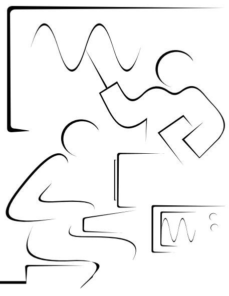
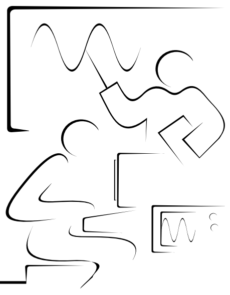
LabVIEW™ Basics II Development Course Manual

Course Software Version 7.0
June 2003 Edition
Part Number 320629L-01



Contents

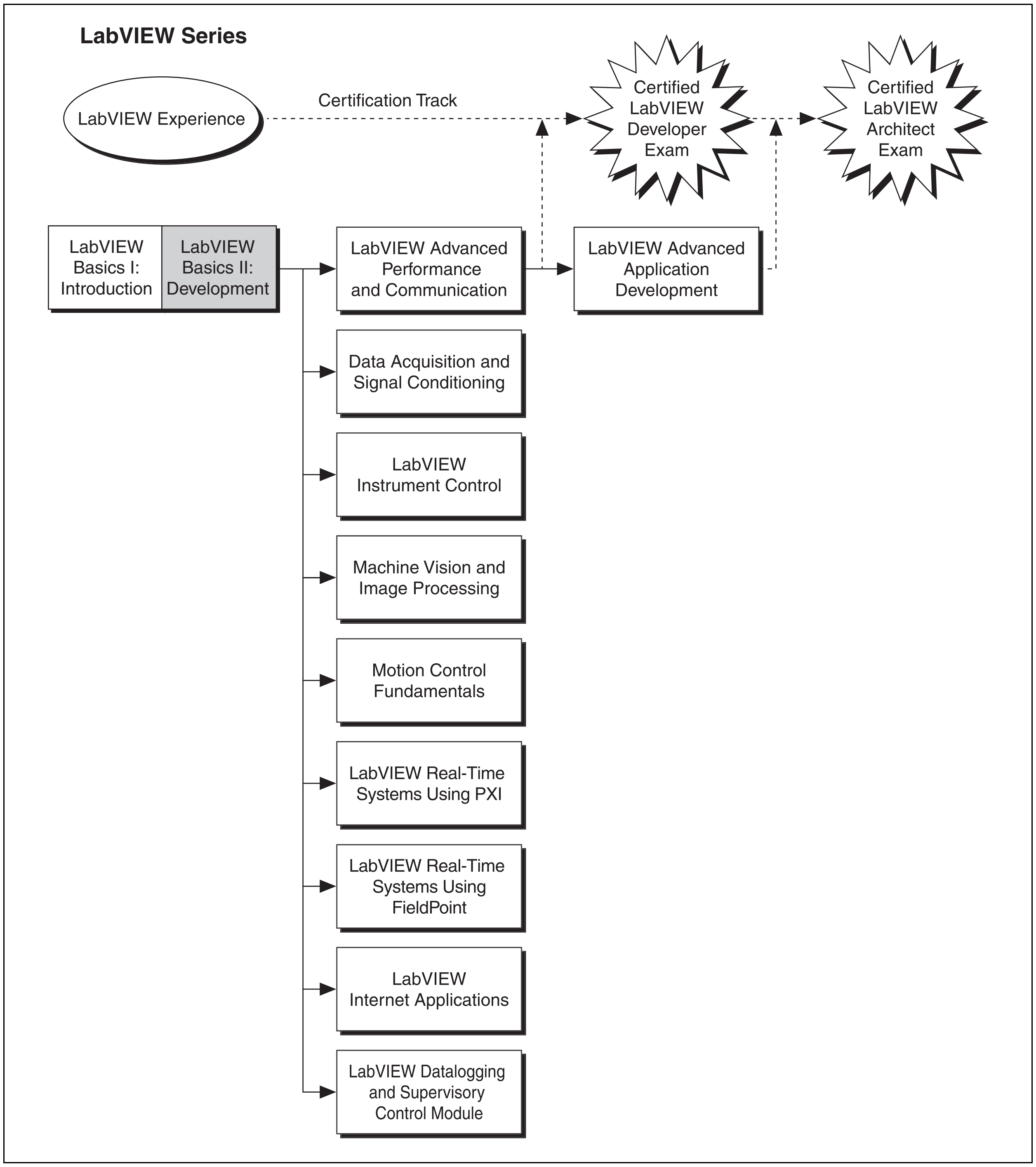
Student Guide



Thank you for purchasing the *LabVIEW Basics II: Development* course kit. You can begin developing an application soon after you complete the exercises in this manual. This course manual and the accompanying software are used in the two-day, hands-on *LabVIEW Basics II: Development* course.

You can apply the full purchase of this course kit toward the corresponding course registration fee if you register within 90 days of purchasing the kit. Visit `ni.com/training` for online course schedules, syllabi, training centers, and class registration.

The *LabVIEW Basics II: Development* course is part of a series of courses designed to build your proficiency with LabVIEW and help you prepare for exams to become an NI Certified LabVIEW Developer and NI Certified LabVIEW Architect. The following illustration shows the courses that are part of the LabVIEW training series. Refer to `ni.com/training` for more information about NI Certification.


LabVIEW Series
LabVIEW Experience
Certification Track
Certified LabVIEW Developer Exam
Certified LabVIEW Architect Exam
LabVIEW Basics I: Introduction
LabVIEW Basics II: Development
LabVIEW Advanced Performance and Communication
LabVIEW Advanced Application Development
Data Acquisition and Signal Conditioning
LabVIEW Instrument Control
Machine Vision and Image Processing
Motion Control Fundamentals
LabVIEW Real-Time Systems Using PXI
LabVIEW Real-Time Systems Using FieldPoint
LabVIEW Internet Applications
LabVIEW Datalogging and Supervisory Control Module

A. About This Manual

This course manual teaches you how to use LabVIEW to develop test and measurement, data acquisition, instrument control, datalogging, measurement analysis, and report generation applications. This course manual assumes that you are familiar with Windows, Macintosh, or UNIX; that you have experience writing algorithms in the form of flowcharts or block diagrams; and that you have taken the *LabVIEW Basics I: Introduction* course or have equivalent experience.

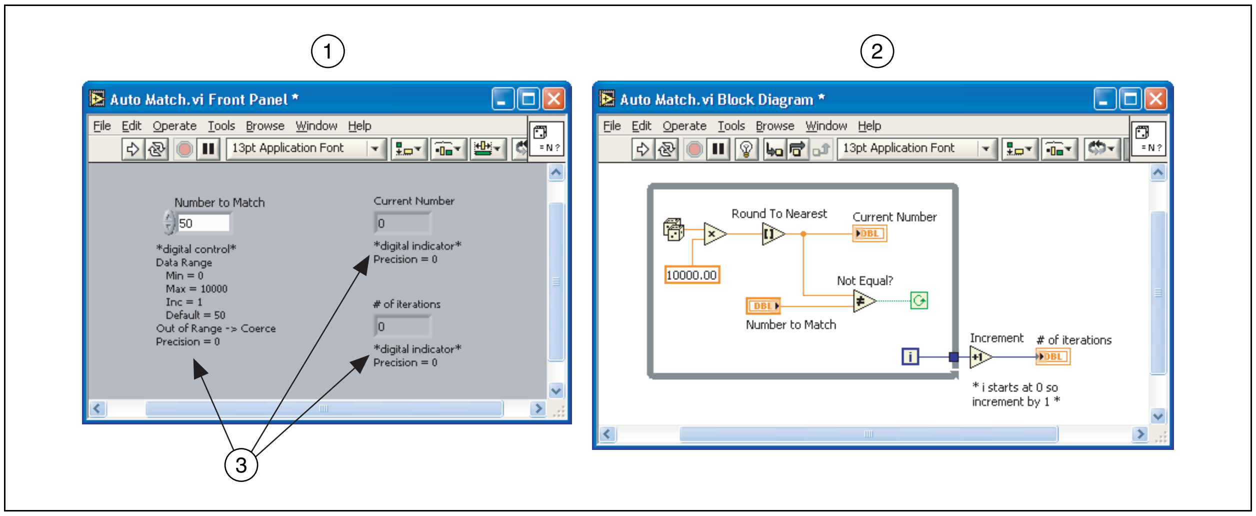
The course manual is divided into lessons, each covering a topic or a set of topics. Each lesson consists of the following:

- An introduction that describes the purpose of the lesson and what you will learn
- A description of the topics in the lesson
- A set of exercises to reinforce those topics
- A set of additional exercises to complete if time permits
- A summary that outlines important concepts and skills taught in the lesson

Several exercises in this manual use a plug-in multifunction data acquisition (DAQ) device connected to a DAQ Signal Accessory containing a temperature sensor, function generator, and LEDs.

Exercises that explicitly require hardware are indicated with an icon, shown at left. You also can substitute other hardware for those previously mentioned. For example, you can use another National Instruments DAQ device connected to a signal source, such as a function generator.

Most exercises show a picture of a *finished* front panel and block diagram after you run the VI, as shown in the following illustration. A description of objects in the block diagram follows each block diagram picture.



1 Front panel	2 Block Diagram	3 *Comments* (do not enter these)

B. What You Need to Get Started

Before you use this course manual, make sure you have all of the following items:

❑ **(Windows)** Windows 98 or later installed on your computer; **(Macintosh)** Power Macintosh running Mac OS 7.6.1 or later; **(UNIX)** Sun workstation running Solaris 2.5 or later and XWindows system software or a PC running Linux kernel 2.0.*x* or later for the Intel *x*86 architecture

❑ **(Windows)** Multifunction DAQ device configured as device 1 using Measurement & Automation Explorer (MAX); **(Macintosh)** Multifunction DAQ device in Slot 1

❑ DAQ Signal Accessory, wires, and cable

❑ LabVIEW Professional Development System 7.0 or later

Note This course assumes you are using the default installation of LabVIEW. If you have changed the palette views from the default settings, some palette paths described in the course may not match your settings. To reset palette views to LabVIEW defaults, select **Tools»Options** and select **Controls/Functions Palettes** from the top pull-down menu. Set **Palette View** to **Express** and set **Format** to **Standard**. Click the **OK** button to apply the changes and close the dialog box.

❑ (Optional) A word processing application such as **(Windows)** Notepad, WordPad, **(Mac OS)** SimpleText, **(UNIX)** Text Editor, vi, or vuepad

❑ *LabVIEW Basics II: Development* course CD, containing the following files

Filename	Description
`Exercises`	Folder containing VIs used in the course
`Solutions`	Folder containing completed course exercises
`Lvb2read.txt`	Text file describing how to install the course software

C. Installing the Course Software

Complete the following steps to install the course software.

Windows

1. Copy the `Exercises` directory to the top level of the `C:\` directory.
2. Copy the `Solutions` directory to the top level of the `C:\` directory.

D. Course Goals and Non-Goals

This course prepares you to do the following:

- Understand the VI development process
- Understand some common VI programming architectures
- Design effective user interfaces (front panels)
- Use data management techniques in VIs
- Use advanced file I/O techniques
- Use LabVIEW to create applications
- Improve memory usage and performance of VIs

You will apply these concepts as you build a project that uses VIs you create throughout the course. While these VIs individually illustrate specific concepts and features in LabVIEW, they constitute part of a larger project you finish in Lesson 5, *Advanced File I/O Techniques*.

The project you build must meet the following criteria:

- Provide a menu-like user interface
- Require the user to log in with a correct name and password
- If the user is not correctly logged in, disable certain features
- Acquire data with the specified user configuration
- Analyze a subset of data and save the results to a file
- Load and view analysis results previously saved to disk

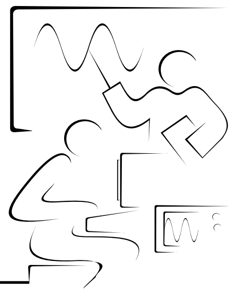
This course does *not* describe any of the following:

- LabVIEW programming methods covered in the *LabVIEW Basics I: Introduction* course
- Every built-in VI, function, or object; refer to the *LabVIEW Help* for more information about LabVIEW features not described in this course
- Developing a complete application for any student in the class; refer to the NI Example Finder, available by selecting **Help»Find Examples**, for example VIs you can use and incorporate into VIs you create

E. Course Conventions

The following conventions appear in this course manual:

»	The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.
💡	This icon denotes a tip, which alerts you to advisory information.
✎	This icon denotes a note, which alerts you to important information.
⚠	This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.
▭	This icon indicates that an exercise requires a plug-in DAQ device.
bold	Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names, controls and buttons on the front panel, dialog boxes, sections of dialog boxes, menu names, and palette names.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.
`monospace`	Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and code excerpts.
`monospace bold`	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.
`monospace italic`	Italic text in this font denotes text that is a placeholder for a word or value that you must supply.
Platform	Text in this font denotes a specific platform and indicates that the text following it applies only to that platform.
right-click	**(Mac OS)** Press <Command>-click to perform the same action as a right-click.



Lesson 1
Planning LabVIEW Applications

This lesson describes some of the issues involved when developing LabVIEW applications, including the design process, the organization of subVI components, and the process of combining those components to create a complete application. This lesson also describes common LabVIEW programming architectures along with some tools to help you build VIs. Additional resources for creating LabVIEW applications are described in the *LabVIEW Development Guidelines* manual.

You Will Learn:

A. Planning and design tips for developing a LabVIEW application

B. How to convert your design outline into LabVIEW subVIs

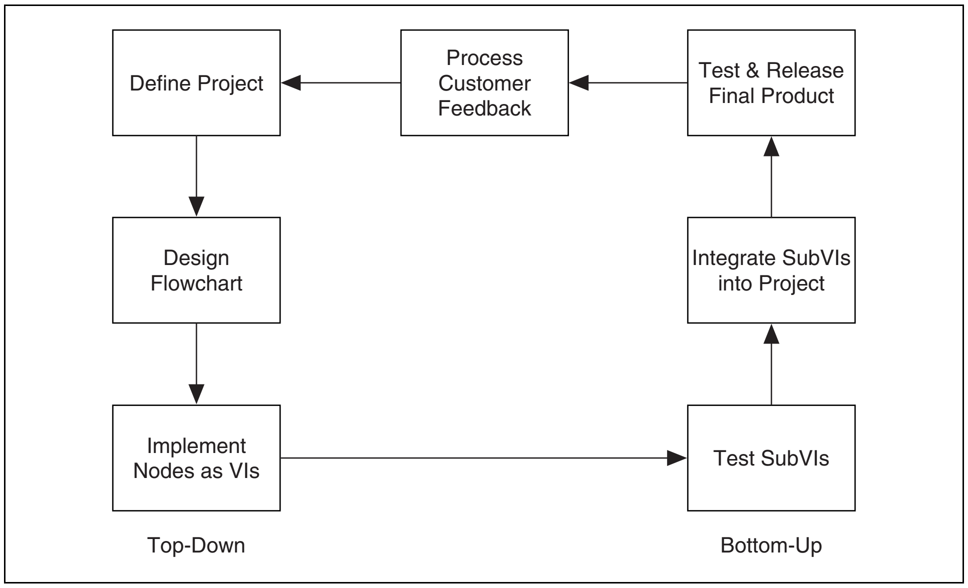
C. Common LabVIEW programming architectures

D. How to create simple user menus

E. About VI templates

A. Planning and Design Process

To design large LabVIEW projects, begin with a top-down approach. First define the general characteristics and specifications of the project. Next, define specific tasks for the general specifications. After defining the tasks the application must perform, develop the subVIs you will assemble to form the completed project. This subVI development represents the bottom-up development period. Customer feedback helps you determine new features and improvements for the next version of the product, bringing you back to the project design phase. The following illustration illustrates this project development process.


Define Project
Process Customer Feedback
Test & Release Final Product
Design Flowchart
Integrate SubVIs into Project
Implement Nodes as VIs
Test SubVIs
Top-Down
Bottom-Up

Designing a flow diagram can help you visualize how your application should operate and how you should set up the overall hierarchy of your project. Because LabVIEW is a dataflow programming language and its block diagrams are similar to typical flowcharts, it is important to carefully plan the flowchart. You can directly implement many nodes of the flowchart as LabVIEW subVIs. By carefully planning the flowchart before implementing your LabVIEW application, you can save development time later. Refer to the *LabVIEW Development Guidelines* manual for more information about developing large applications.

When designing large projects, remember the following development guidelines:

- Accurately define the system requirements.
- Clearly determine expectations of the end user.
- Document what the application must accomplish.
- Plan for future modifications and additions.

B. Implementation Process

After completing the planning process, implement your application by developing subVIs that correspond to flowchart nodes. Although you cannot always use this modular approach, it helps in the development of your application. By clearly defining a hierarchy of the application requirements, you create a blueprint for the organization of the VIs you develop.

Modular development also makes it much easier for you to test small portions of an application and later combine them. If you build an entire application on one block diagram without subVIs, you might not be able to start testing until you have developed the majority of the application. Once you have developed a major portion of an application, it is very cumbersome to debug problems that might arise. By testing smaller, more specific VIs, you can determine initial design flaws and correct them before investing hours of implementation time.

By planning modular, hierarchical development, it is easier to maintain control of the source code for your project and have a better knowledge of the project status. Another advantage of using subVIs is that future modifications and improvements to the application are much easier to implement.

After building and testing the necessary subVIs, use them to complete your LabVIEW application. This is referred to as bottom-up development.

C. LabVIEW Programming Architectures

You can develop better programs in LabVIEW and in other programming languages if you follow consistent programming techniques and architectures. Structured programs are easier to maintain and understand. Structured programs are modular, which means you can reuse the code in different operations. Structured programs also are easy to understand by looking at the code itself. This also makes the code easier to edit

Another good programming practice is to document the code to explain what the code does. This helps prevent accidental removal of code, which could cause errors.

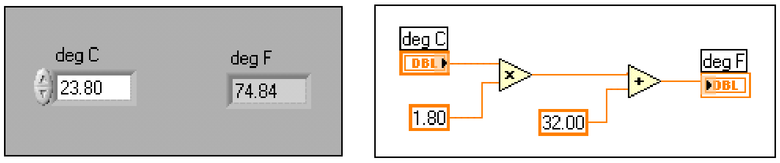
One of the best ways to create an easy-to-understand program architecture is to f make subVIs reusable for similarly grouped operations. When you also document the code, a modular VI is easy to understand and modify in the future.

This section describes some of the common types of VI architectures such as simple, general, parallel loops, multiple Case structures, and state machines.

Simple VI Architecture

When performing calculations or making quick lab measurements, you do not need a complicated architecture. Your program might consist of a single VI that takes a measurement, performs calculations, and either displays the results or records them to disk. The simple VI architecture usually does not require a specific start or stop action from the user. The user just clicks the **Run** button. Use this architecture for simple applications or for functional components within larger applications. You can convert these simple VIs into subVIs that you use as building blocks for larger applications.

The following figure displays the front panel and block diagram of the Convert C to F VI built in the *LabVIEW Basics I: Introduction* course. This VI performs a single task: it converts a value in degrees Celsius to degrees Fahrenheit. You can use this VI as a subVI whenever you need to convert Celsius to Fahrenheit, instead of rebuilding the block diagram every time you need to perform the conversion.

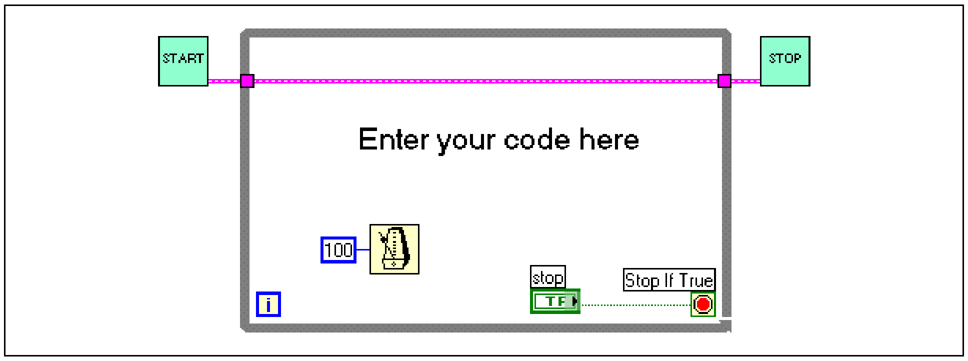


General VI Architecture

In designing an application, you generally have the following three main phases:

Startup	This phase initializes hardware, reads configuration information from files, or prompts the user for data file locations.
Main Application	This phase consists of at least one loop that repeats until the user decides to exit the program, or the program terminates for other reasons such as I/O completion.
Shutdown	This phase closes files, writes configuration information to disk, or resets I/O to the default state.

The following block diagram shows this general VI architecture.

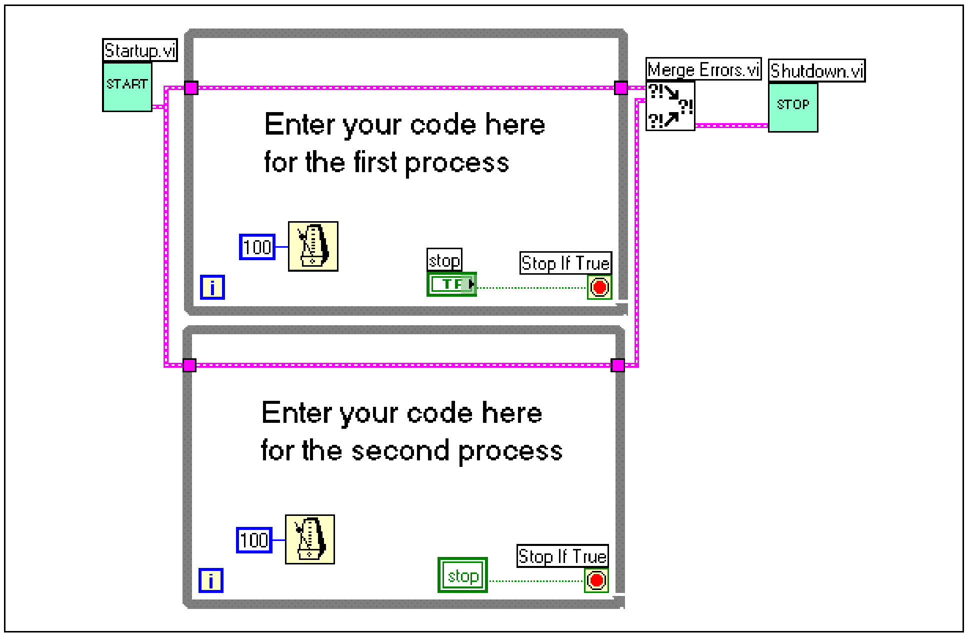


In the previous block diagram, the error cluster wires control the execution order of the three sections. The While Loop does not execute until the Start VI finishes running and returns the error cluster. Consequently, the Stop VI cannot run until the main program in the While Loop finishes and the error cluster data leave the loop. The Wait function is required in most loops, especially if that loop is monitoring user input on the front panel. Without the Wait function, the loop might run continuously, which might use all of the computer system resources. The Wait function forces the loop to run asynchronously even if you specify 0 milliseconds as the wait period. If the operations inside the main loop react to user inputs, then you can increase the wait period to a level acceptable for reaction times. A wait of 100–200 ms is usually good because most users will not detect that amount of delay between clicking a button on the front panel and the subsequent event executing.

For simple applications, the main application loop is obvious and easy to understand. When you have complicated user interfaces or multiple tasks such as user actions, I/O triggers, and so on, the main application phase gets more complicated.

Parallel Loop VI Architecture

Some applications require the program to respond to and run several tasks concurrently. One way of designing the main section of this application is to assign a different loop to each task. For example, you might have a different loop for each button on the front panel and for every other kind of task, such as a menu selection, I/O trigger, and so on. The following block diagram shows this parallel loop VI architecture.

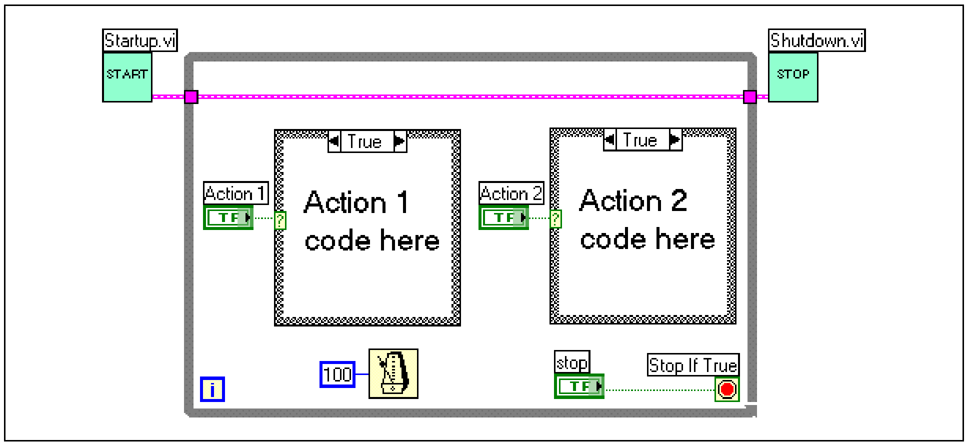

Startup.vi
START
Enter your code here for the first process
100
stop
Stop If True
Merge Errors.vi
Shutdown.vi
STOP
Enter your code here for the second process
100
stop
Stop If True

This structure is straightforward and appropriate for some simple menu type VIs where you expect a user to select from one of several buttons that perform different actions. The parallel loop VI architecture lets you handle multiple, simultaneous, independent processes. In this architecture, responding to one action does not prevent the VI from responding to another action. For example, if a user clicks a button that displays a dialog box, parallel loops can continue to respond to I/O tasks.

However, the parallel loop VI architecture requires you to coordinate and communicate among different loops. The **Stop** button for the second loop in the previous block diagram is a local variable. You cannot use wires to pass data between loops because doing so prevents the loops from running in parallel. Instead, you must use a messaging technique for passing information among processes. This can lead to race conditions where multiple tasks attempt to read and modify the same data simultaneously resulting in inconsistent behavior that is difficult to debug. Refer to Lesson 4, *Local and Global Variables*, for more information about local variables, global variables, and race conditions.

Multiple Case Structure VI Architecture

The following block diagram shows a VI that handles multiple tasks and passes data back and forth between each task. Instead of using multiple loops, you can use a single loop that contains separate Case structures to handle each task. Use this VI architecture when you have several buttons on the front panel that initiate different tasks.



The multiple case structure VI architecture lets you use wires to pass data. This improves readability and reduces the need for using global variables. This architecture also makes race conditions less likely. You also can use Shift Registers on the loop border to remember values from one iteration to the next and to pass data.

With this architecture you can end up with block diagrams that are very large and, consequently hard to read, edit, and debug. Also, because all Case structures are in the same loop, each one is handled serially. If one task takes a long time, the loop cannot handle other tasks. A related problem is that all tasks are handled at the same rate because no task can repeat until all objects in the loop complete. In some applications, you might want to set the priority of user interface actions to be fairly low compared to I/O tasks.

State Machine VI Architecture

The state machine VI architecture uses only one Case structure, allowing VIs to be more compact. The current condition determines the function the VI executes. Use this architecture for VIs that are easily split into several simpler tasks, such as VIs that act as a user interface.

A state machine in LabVIEW consists of a While Loop, a Case structure, and a Shift Register. Each state of the state machine is a separate case in the

Case structure. You place VIs and other code that the state should execute within the appropriate case. A Shift Register stores the state to be executed upon the next iteration of the loop. The block diagram of a state machine VI with five states, or cases, appears in the following figures.

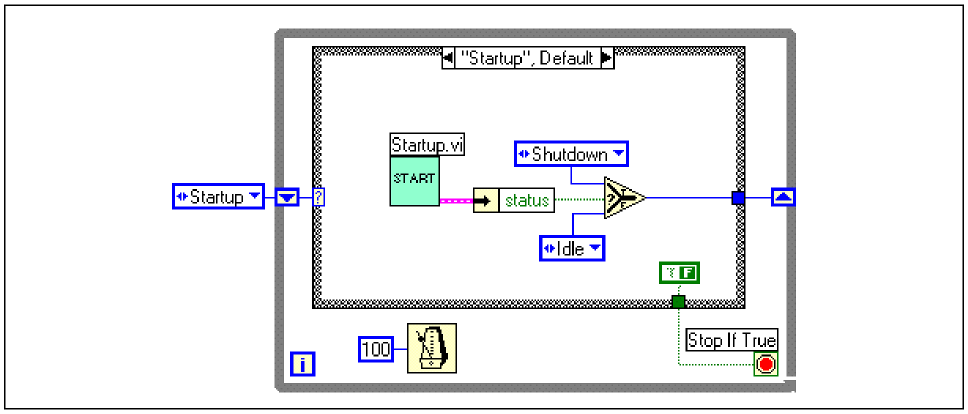


Figure 1-1. State Machine with Default Startup State

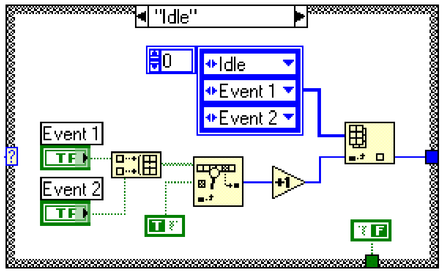


Figure 1-2. Idle State

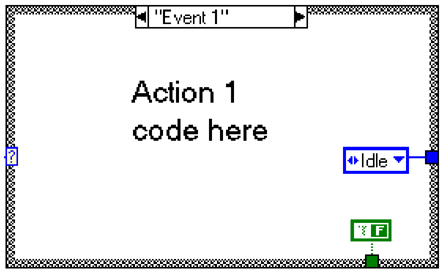


Figure 1-3. Event 1 State

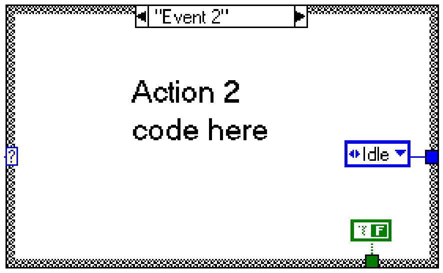


Figure 1-4. Event 2 State

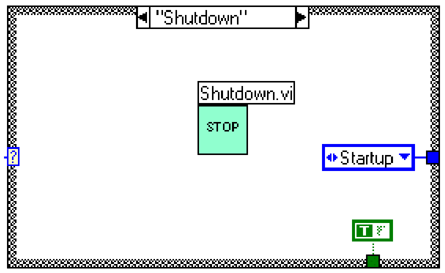


Figure 1-5. Shutdown State

In the state machine VI architecture, you design the list of possible events, or states, and then map them to each case. For the VI in the previous example, the possible states are Startup, Idle, Event 1, Event 2, and Shutdown. An enumerated constant stores the states. Each state has its own case in the Case structure. The outcome of one case determines the case that next executes. The Shift Register stores a value that determines which case runs next. If an error occurs in any of the states, the Shutdown case is called.

State machine VI architecture makes the block diagram much smaller, and therefore, easier to read and debug. Another advantage of state machine architecture is that each case determines what the next state will be as it runs, unlike sequence structures that cannot skip a frame.

A disadvantage of the state machine VI architecture is that with the approach in the previous example, it is possible to skip states. If two states in the structure are called at the same time, this model handles only one state, and the other state does not execute. This can lead to errors that are difficult to debug because they are difficult to reproduce. More complex versions of the state machine VI architecture contain extra code that builds a queue of events, or states, so that you do not miss a state. Event queues are discussed in more detail in the *LabVIEW Advanced: Performance and Communication* course.

More About Programming Architecture

As with other programming languages, you can use different methods and programming techniques to design a VI in LabVIEW. This section describes some of the common methods and VI architectures you can use for creating VIs.

VI structures become more complex when applications perform more tasks and combine different hardware types, user interface requirements, and error checking methods. However, you can use the same basic programming architectures. Examine the larger examples and demos that ship with LabVIEW and write down which common VI architectures are used and why. Refer to the *LabVIEW Development Guidelines* manual for additional resources for creating LabVIEW applications.

D. Creating a Simple User Menu

One of the simplest user menus to create is a cluster of Boolean objects. The Boolean objects in the cluster require specific mechanical action settings in order to behave correctly.

Mechanical Action of Boolean Objects

Boolean controls have six types of mechanical action that allow you to customize Boolean objects to create front panels that more closely resemble the behavior of physical instruments. You can set the mechanical action of a Boolean control by right-clicking the control and selecting **Properties** from the shortcut menu to display the **Boolean Properties** dialog box for the control. The dialog box includes a **Button behavior** list, a **Behavior Explanation** section, and a **Preview Selected Behavior** section.

Note In the icons that appear in the **Behavior Explanation** section, M represents the motion of the mouse button when you operate the control, V represents the output value of the control, and RD represents the point in time the VI reads the control.

You can select from the following button behaviors:

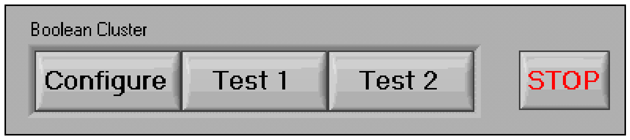
- **Switch When Pressed**—Changes the control value each time you click it with the Operating tool, similar to a light switch. How often the VI reads the control does not affect this action.
- **Switch When Released**—Changes the control value only after you release the mouse button during a mouse click within the graphical boundary of the control. How often the VI reads the control does not affect this action.
- **Switch Until Released**—Changes the control value when you click it and retains the new value until you release the mouse button. At this time, the control reverts to its original value, similar to the operation of a door buzzer. How often the VI reads the control does not affect this action.
- **Latch When Pressed**—Changes the control value when you click it and retains the new value until the VI reads it once. At this point the control reverts to its default value, even if you keep pressing the mouse button. This action is similar to a circuit breaker and is useful for stopping While Loops or for getting the VI to perform an action only once each time you set the control.

- **Latch When Released**—Changes the control value only after you release the mouse button within the graphical boundary of the control. When the VI reads it once, the control reverts to the old value. This action guarantees at least one new value. This action is similar to dialog box buttons and system buttons.
- **Latch Until Released**—Changes the control value when you click it and retains the value until the VI reads it once or until you release the mouse button, depending on which one occurs last.

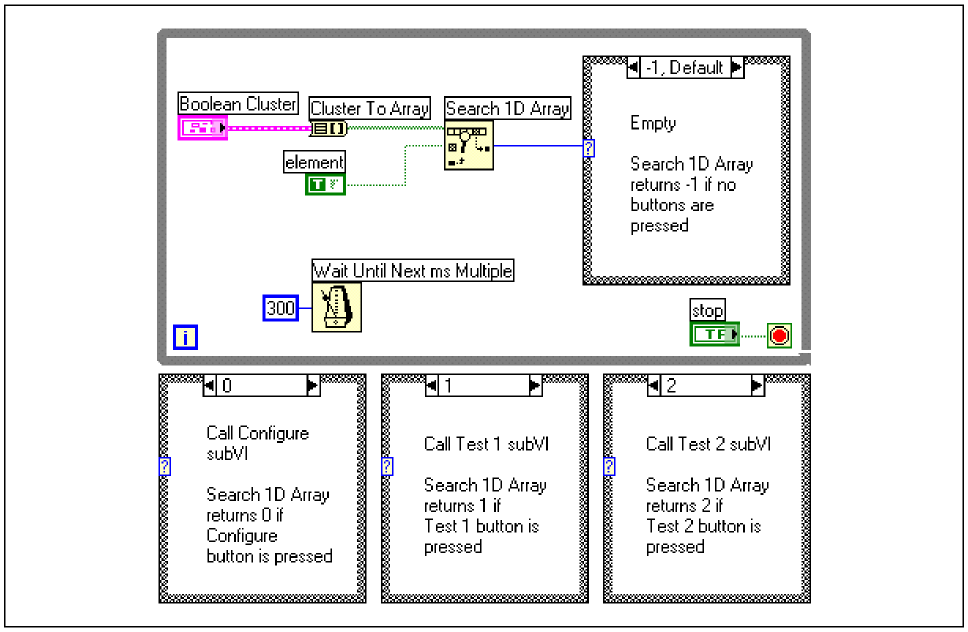
Note You cannot use any latch action for objects with a local variable because the first local variable to read a Boolean control with latch action would reset its value to the default.

Using Boolean Clusters as Menus

You can use latched Boolean buttons in a cluster to build a menu for a state machine application. For example, consider an application where the operator configures a system and runs one of two tests. A possible menu VI for this application is shown in the following illustration. The mechanical action is set to **Latch When Released** for the cluster of buttons.



The Cluster to Array function converts the Boolean cluster to a Boolean array with three elements. That is, each button in the cluster represents an element in the array. The Search 1D Array function on the **Functions»All Functions»Array** palette searches the 1D array of Boolean values created by the Cluster to Array function for a value of TRUE. A TRUE value for any element in the array indicates the user clicked a button in the cluster. The Search 1D Array function returns the index of the first TRUE value it finds in the array and passes that index value to the selector terminal of a Case structure, as shown in the following block diagram.


Boolean Cluster
Cluster To Array
Search 1D Array
element
-1, Default
Empty
Search 1D Array returns -1 if no buttons are pressed
Wait Until Next ms Multiple
300
stop
0
Call Configure subVI
Search 1D Array returns 0 if Configure button is pressed
1
Call Test 1 subVI
Search 1D Array returns 1 if Test 1 button is pressed
2
Call Test 2 subVI
Search 1D Array returns 2 if Test 2 button is pressed

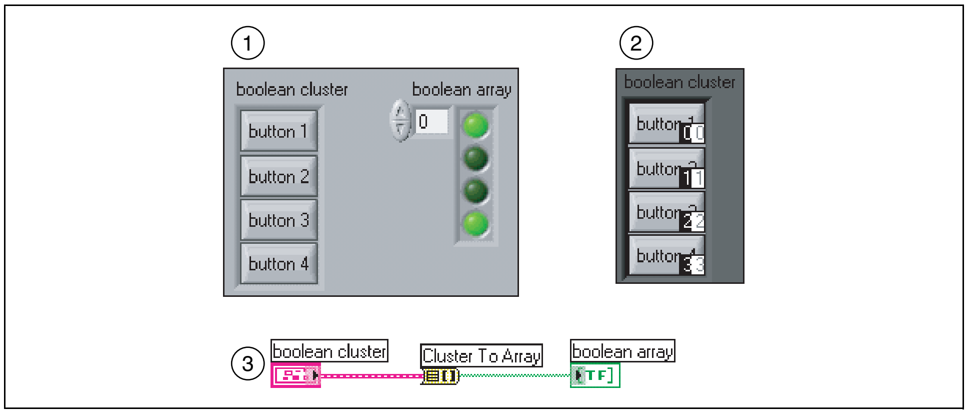
If you do not click a button, Search 1D Array returns an index value of `-1` and the `-1` case executes, which does nothing. Clicking the **Configure** button executes the `0` case, which calls a configure subVI, clicking the **Test 1** button executes the `1` case, which calls the first test subVI, and clicking the **Test 2** button executes the `2` case, which calls the second test subVI. The While Loop repeatedly checks the state of the Boolean cluster control until you click the **Stop** button.

Converting Clusters to Arrays

You can convert a cluster to an array if all cluster elements have the same data type, such as the Boolean data type. After converting cluster elements to an array, you can use the Array functions to process the cluster elements.

The Cluster to Array function located on the **Functions»All Functions»Cluster** and **Functions»All Functions»Array** palettes converts a cluster of elements of the same type to a 1D array of elements of the same type.

The following illustration shows a four-component Boolean cluster converted to a four-element Boolean array. The index of each element in the array corresponds to the logical order of the component in the cluster. For example, Button 1 (component 0) corresponds to the first element (index 0) in the array, Button 2 (component 1) corresponds to the second element (index 1), and so on.



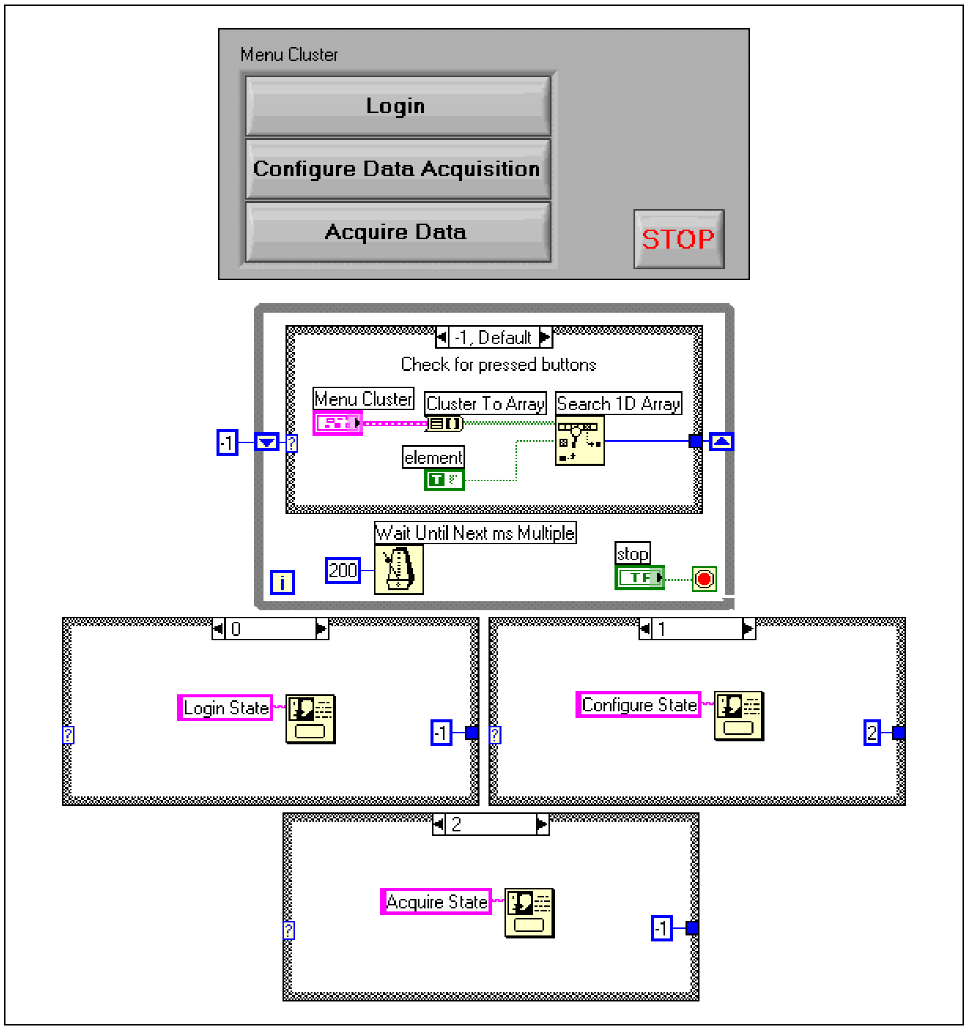
1 Front Panel   2 Cluster Panel   3 Block Diagram

The Array to Cluster function, located on the **Functions»All Functions»Cluster** and **Functions»All Functions»Array** palettes, converts a 1D array to a cluster of elements of the same type as the array elements.

You can combine the concept of a state machine with a Boolean menu cluster to provide a powerful system for menus. For example, if you need to develop an application to login a user, configure an acquisition, and acquire data, first divide it into a series of states, as shown in the following table.

State Value	State Name	Description	Next State
–1, Default	No Event	Monitor Boolean menu to determine the next state	Depends on the Boolean button pressed. If no button is pressed, next state is No Event.
0	Login	Log in user	No Event (–1)
1	Configure	Configure acquisition	Acquire (2)
2	Acquire	Acquire data	No Event (–1)

The following figure shows the state machine for this application.



The front panel consists of a Boolean button cluster where each button triggers a state in the state machine. In state –1 (the No Event state), the Boolean button cluster is checked to see if a button has been pressed. The Search 1D Array function returns the index of the button pressed, or –1 if no button is pressed, to determine the next state to execute. That state value is loaded into the Shift Register, so that on the next iteration of the While Loop the selected state executes.

In each of the other states, the Shift Register is loaded with the next state to execute using a numeric constant. Normally this is state –1, so that the Boolean menu is checked again, but in state 1 (Configure) the subsequent state is state 2 (Acquire).

Exercise 1-1 State Machine Menu VI

Objective: To build the menu system for the sample application.

A set of dependencies exist between the different operations of the application you build in this course. Under most circumstances, after performing a certain action, the application should return to a No Event state, in which the application monitors a menu to determine which button the user pressed.

The dependencies of the application can be described as a simple state machine, where each numeric state leads to another subsequent state. The following table summarizes this series of dependencies.

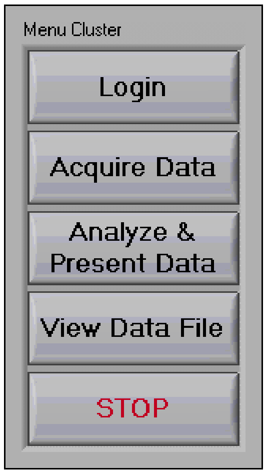
State Value	State Name	Description	Next State
–1, Default	No Event	Monitor Boolean menu to determine the next state	Depends on the Boolean button pressed. If no button is pressed, the next state is No Event.
0	Login	Log in user	No Event
1	Acquire	Acquire data	No Event
2	Analyze	Analyze data, possibly save to file	No Event
3	View	View saved data files	No Event
4	Stop	Stop VI	No Event

In this exercise, you build the state machine to be used in this application and observe its operation.

Front Panel

1. Open a new VI.
2. Place a Cluster control, located on the **Controls»All Controls»Array & Cluster** palette, on the front panel.
3. Place any Boolean button, located on the **Controls»Buttons & Switches** palette, in the cluster.

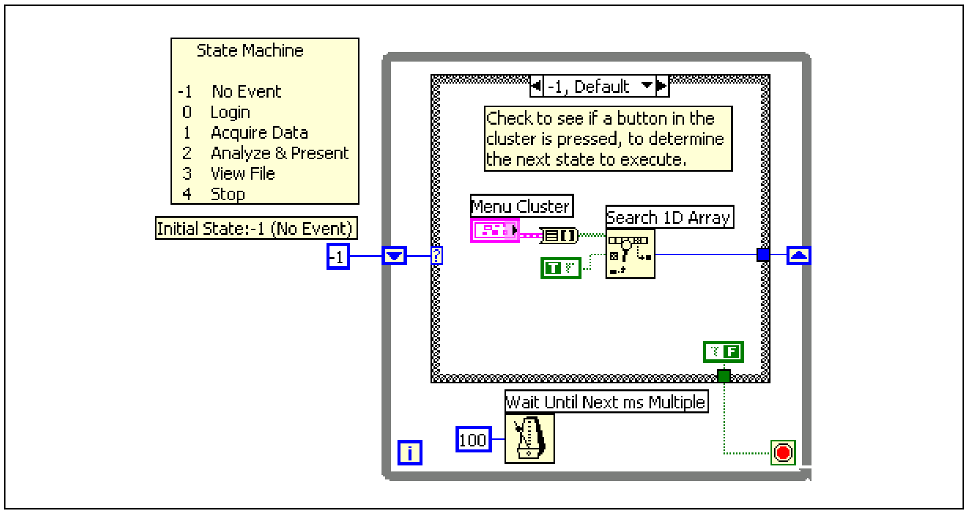
4. Set the mechanical action of the button to **Latch When Released**.
   a. Right-click the Boolean control and select **Properties** from the shortcut menu to display the **Operation** page of the **Boolean Properties** dialog box.
   b. Select **Latch When Released** from the **Button behavior** list. An explanation of the behavior appears in the **Behavior Explanation** section and you can test the selected behavior in the **Preview Selected Behavior** section.
   c. Click the **OK** button to apply the changes and close the dialog box.



5. Use the Boolean button to create the remaining buttons in the cluster.
   a. Use a 20-point, bold, Application font to label the buttons. Enter the text for the button with the largest label first.
   b. Use **Copy** and **Paste** to create the other buttons.
   c. Use the **Align Objects** and **Distribute Objects** toolbar buttons to arrange the buttons.
6. Right-click the edge of the cluster and select **AutoSizing»Size to fit** from the shortcut menu.
7. The user clicks a button in the cluster to trigger the appropriate state in the state machine. Right-click the edge of the cluster and select **Reorder Controls In Cluster** to verify the cluster order. Make sure that the **Login** button is at cluster order 0. **Acquire Data** is cluster order 1, **Analyze & Present Data** is cluster order 2, **View Data File** is cluster order 3, and **Stop** is cluster order 4. The cluster order of the menu cluster determines which numeric state executes. Click the **OK** button to confirm the cluster order.

Block Diagram

8. Press the <Ctrl-E> keys or select **Window»Show Block Diagram** to open the block diagram. Build the following block diagram.


State Machine
-1 No Event
0 Login
1 Acquire Data
2 Analyze & Present
3 View File
4 Stop
Initial State:-1 (No Event)
-1, Default
Check to see if a button in the cluster is pressed, to determine the next state to execute.
Menu Cluster
Search 1D Array
Wait Until Next ms Multiple
100

a. Place a While Loop, located on the **Functions»Execution Control** palette, on the block diagram. This structures the VI to continue to generate and analyze data until the user clicks the **Stop** button. Create the Shift Register by right-clicking the left or right border of the loop and selecting **Add Shift Register** from the shortcut menu.

b. Place an Enumerated Type Constant, located on the **Functions»All Functions»Numeric** palette to the left of the While Loop. Right-click the constant and select **Change to Control** from the shortcut menu. Enter the state values from the previous table in the Enumerated Type Control. Wire the control to the Shift Register.

c. Place a Case structure, located on the **Functions»Execution Control** palette, in the While Loop. This structure creates the states for the state machine. Wire the Shift Register to the Case Selector input. Right-click the border of the Case structure and select **Add Case After** from the shortcut menu four times to create four additional cases. Select the default case and complete steps d–g.

d. Place the Wait Until Next ms Multiple function, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. This function causes the While Loop to execute 10 times a second. Create the constant by right-clicking the input terminal and selecting **Create»Constant** from the shortcut menu.

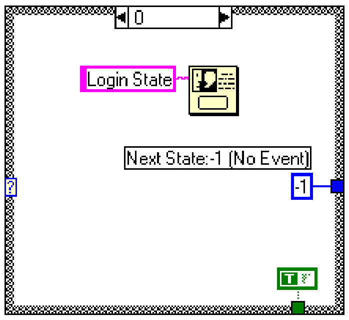
e. Place the Cluster To Array function, located on the **Functions»All Functions»Cluster** or **Functions»All Functions»Array** palette, on the block diagram. In this exercise, this function converts the cluster of Boolean buttons into an array of Boolean data types. The Boolean object at cluster order 0 becomes the Boolean element at array index 0, cluster order 1 becomes array index 1, and so on.

f. Place the Search 1D Array function, located on the **Functions»All Functions»Array** palette, on the block diagram.

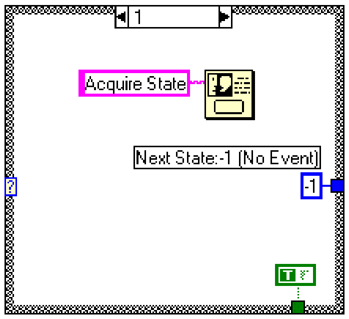
g. Place a True constant, located on the **Functions»Arithmetic & Comparison»Express Boolean** palette, on the block diagram. Wire the True constant to the **element** input of the Search 1D Array function. This causes the function to search for a TRUE value in the Boolean array that Cluster to Array returns. A TRUE value for any element indicates that the corresponding button was clicked. The function returns a value of –1 if you do not click a button.

Complete cases 0–4 in the Case structure so they will indicate which state has been selected and loaded into the Shift Register when the user clicks the corresponding button.

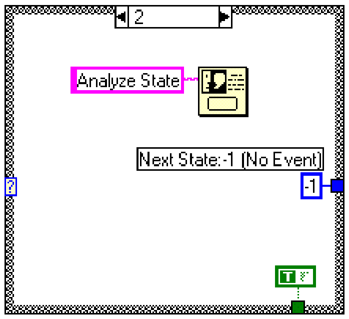
h. Select case 0 of the Case structure and place the One Button Dialog function, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. Right-click the One Button Dialog function and select **Create Constant** from the shortcut menu to create the string constant. Enter the text shown in the following figure in the constant. Wire a `-1` constant to the Case structure tunnel.



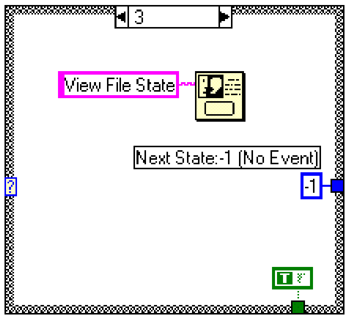
i. Select case 1 of the Case structure and place the One Button Dialog function, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. Right-click the One Button Dialog function and select **Create Constant** from the shortcut menu to create the string constant. Enter the text shown in the following figure in the constant. Wire a `-1` constant to the Case structure tunnel.



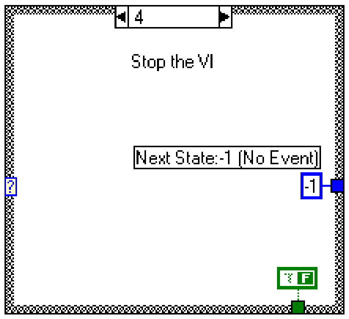
j. Select case 2 of the Case structure and place the One Button Dialog function, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. Right-click the One Button Dialog function and select **Create Constant** from the shortcut menu to create the string constant. Enter the text shown in the following figure in the constant. Wire a `-1` constant to the Case structure tunnel.



k. Select case 3 of the Case structure and place the One Button Dialog function, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. Right-click the One Button Dialog function and select **Create Constant** from the shortcut menu to create the string constant. Enter the text shown in the following figure in the constant. Wire a `-1` constant to the Case structure tunnel.



l. Select case 4 of the Case structure. Wire a `-1` constant to the Case structure tunnel and complete the case as shown in the following figure.



9. Select **File»Save As** to save the VI as `Menu.vi` in the `C:\Exercises\LabVIEW Basics II` directory. You will use this VI later in the course.
10. Display the front panel and run the VI. When you click the **Login**, **Acquire Data**, **Analyze & Present Data**, or **View Data File** buttons, a dialog box appears to indicate that you are in the associated state.
11. Using the Single-Step and Execution Highlighting features, observe how the VI executes. Notice that until you click a button, the Search 1D Array function returns a value of –1, which causes the While Loop to continuously execute state –1. Once you click a button, the index of the Boolean data types is used to determine the next state to execute. Notice how the states of the VI correspond to the states in the table at the beginning of this exercise.

    In later exercises, you substitute subVIs that you create for the One Button Dialog functions to build the application.
12. Click the **Stop** button on the front panel to halt execution.
13. Close the VI when you are finished.

End of Exercise 1-1

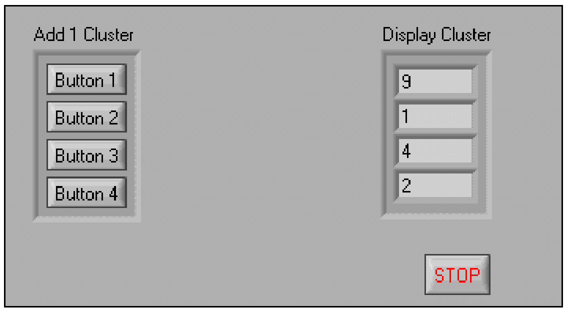
Exercise 1-2 Cluster Conversion Example VI (Optional)

Objective: To examine a VI that uses clusters to process data.

Examine a VI that uses clusters to process data. The VI features a cluster containing four labeled buttons. The VI keeps track of the number of times you click each button.

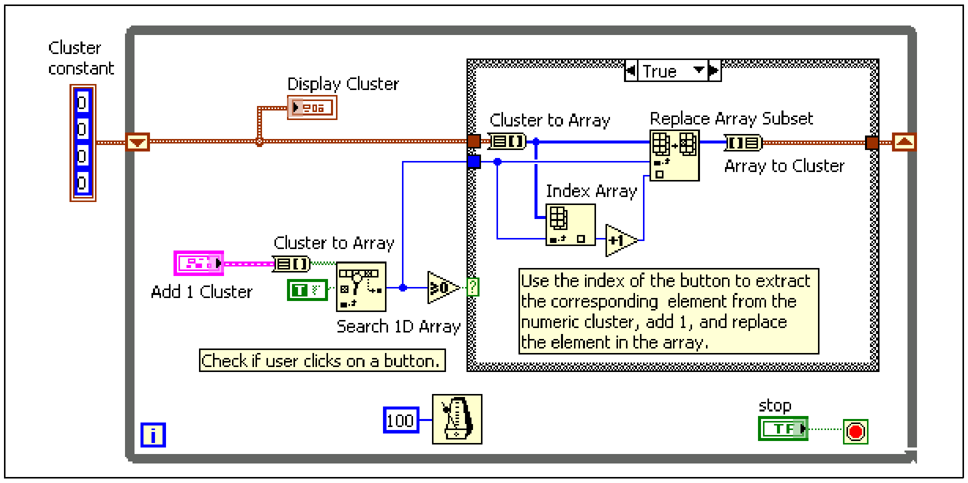
Front Panel

1. Open the Cluster Conversion Example VI located in the `C:\Exercises\LabVIEW Basics II` directory.



Block Diagram

2. Display and examine the block diagram. The FALSE case is empty except for passing the cluster from the left Shift Register to the right Shift Register.



Use the index of the button to extract the corresponding element from the numeric cluster, add 1, and replace the element in the array.

Check if user clicks on a button.

3. Run the VI. Click the buttons on the front panel. The corresponding numeric indicator should increment each time you click a button.
4. Close the VI. Do not save any changes.

End of Exercise 1-2

E. LabVIEW Template VIs

The LabVIEW template VIs include the subVIs, functions, structures, and front panel objects you need to get started building common measurement applications. Template VIs open as untitled VIs that you must save. Select **File»New** to display the **New** dialog box, which includes the template VIs. You also can display the **New** dialog box by clicking the **New** button on the **LabVIEW** dialog box.

You can create custom template VIs to avoid placing the same components on the front panel or block diagram each time you want to perform a similar operation. Save a VI you create as a template by selecting **File»Save as** to display a file dialog box. Select **Template VIs** from the **Save as type** pull-down menu. Enter the name of the template VI and click the **Save** button.

Caution Do *not* save your own template VIs in the `vi.lib` directory because LabVIEW overwrites these files when you upgrade or reinstall.

Exercise 1-3 Single Loop Application and Standard State Machine Template VIs

Objective: To examine two of the template VIs included with LabVIEW.

Examine two template VIs that show both the general VI architecture and the state machine VI architecture.

Single Loop Application Template VI Front Panel

1. Click the **New** button on the **LabVIEW** dialog box to open the **New** dialog box. In the **Create new** section, select **VI from Template»Frameworks»Single Loop Application** and click the **OK** button to open a VI with a front panel that contains only a **Stop** button.

Single Loop Application Template VI Block Diagram

2. Open and examine the block diagram.

   The Single Loop Application VI template uses the general VI architecture. It contains a While Loop that stops when you click the button on the front panel. The Time Delay Express VI in the loop ensures that this loop does not use all the system resources.

3. Return to the front panel and run the VI. It does nothing but continues to run until you click the **Stop** button.
4. Stop and close this VI when you are finished. Do not save the VI.

Standard State Machine Template VI Front Panel

5. Click the **New** button on the **LabVIEW** dialog box to open the **New** dialog box. In the **Create new** section, select **VI from Template»Frameworks»Design Patterns»Standard State Machine** and click the **OK** button. The front panel for this template is empty.

Standard State Machine Template VI Block Diagram

6. Open and examine the block diagram.

   The Standard State Machine VI template uses a Case structure within a While Loop where each case of the Case structure is a different state of the overall application. The next state is determined while the VI is running based on what happens in the current state. This example uses an enumerations that are type definitions. Refer to Lesson 3, *Object Properties*, for more information about type definitions for controls.

7. Close this VI when you are finished. Do not save the VI.

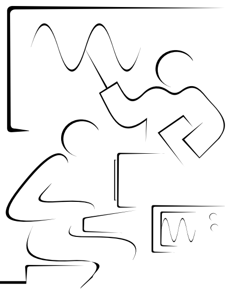
End of Exercise 1-3

Summary

- Use a top-down approach to plan the overall strategy for a project.
- Use a bottom-up approach to develop and implement an application.
- When designing a LabVIEW application, it is important to determine the end-user's expectation, exactly what the application must accomplish, and what future modification might be necessary before you invest a great deal of time developing subVIs. You should design a flowchart to help you understand how the application should operate and discuss this in detail with your customer.
- After you design a flowchart, you can develop VIs to accomplish the various steps in your flowchart. It is a good idea to modularize your application into logical subtasks when possible. Working with small modules allows you to debug an application by testing each module individually. This approach also makes it much easier to modify the application in the future.
- The most common VI architectures are the simple, general, parallel loop, multiple Case structure, and state machine. The architecture you select depends on what you want your application to do.
- The state machine VI architecture is useful for user interface VIs, and it enables you to generate clean, simple code. For many applications, state machines provide a scalable, readable VI with a minimal number of nested structures and variables.
- Latched Boolean buttons in a cluster can be used to build menus for applications.
- You can convert a cluster containing components of the same data type to an array and then use the Array functions to process the cluster components. The Cluster To Array function, located on the **Functions»All Functions»Cluster** or **Functions»All Functions»Array** palette, converts a cluster to a 1D array.
- LabVIEW includes VI templates to help you get started building common measurement applications.
- Refer to the *LabVIEW Development Guidelines* manual for more information about designing and building VIs.

Notes

Notes



Lesson 2
VI Design Techniques

This lesson describes good techniques for designing and building VIs. The techniques discussed include an overview of user interface design issues, good block diagram construction and wiring techniques, error handling, and an overview of LabVIEW run-time menus. All these techniques are important as you design and build more involved and complex VIs.

You Will Learn:

A. Basic user interface issues

B. Dataflow concepts

C. Guidelines for proper wiring

D. About block diagram comments

E. Block diagram checklist

F. Error handling techniques

G. Hierarchical design techniques

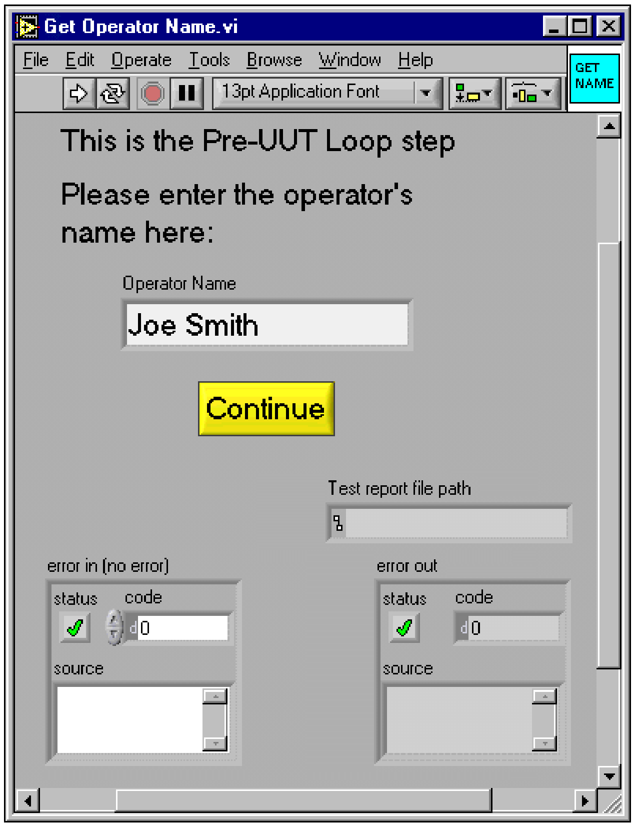
H. Sequence structures

I. LabVIEW run-time menus

A. User Interface Design

When you develop applications for other people to use, you need to follow some basic rules regarding the user interface. For example, if the front panel contains too many objects or has a distracting mixture of color and text, the users might not use the VI properly or might miss important information in the data.

One rule to follow when building a user interface for a VI is to show only items on the front panel that the user needs to see at that time. The following front panel shows a subVI that prompts the user for a login name. It has error clusters because it is a subVI, but the user does not need to see those items or the path name of the file.



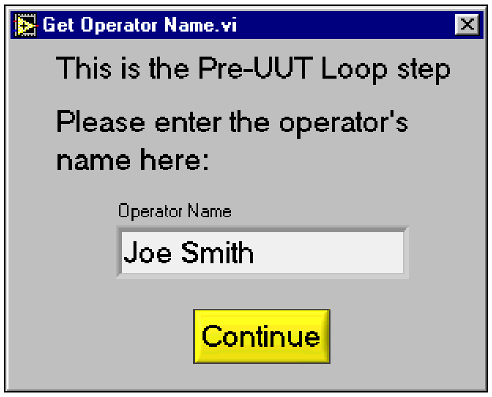
This is the Pre-UUT Loop step

Please enter the operator's name here:

Operator Name

Joe Smith

The following example shows the same subVI after resizing the front panel and removing the menu, scrollbars, and toolbar using the **File»VI Properties»Window Appearance»Dialog** option.



Get Operator Name.vi

This is the Pre-UUT Loop step

Please enter the operator's name here:

Operator Name

Joe Smith

Using Color

Proper use of color can improve the appearance and functionality of your front panel. Using too many colors, however, can result in color clashes that cause the front panels to look too busy and distracting.

LabVIEW provides a color picker that can aid in selecting appropriate colors. Select the Coloring tool and right-click an object or workspace to display the color picker. The top of the color picker contains a grayscale spectrum and a box you can use to create transparent objects. The second spectrum contains muted colors that are well suited to backgrounds and front panel objects. The third spectrum contains colors that are well suited for highlights. Moving your cursor vertically from the background colors to the highlight colors helps you select appropriate highlight colors for a specific background color.

The following tips are helpful for color matching:

- Use the default LabVIEW colors. If a color is not available on a computer, LabVIEW replaces it with the closest match.
- Start with a gray scheme. Select one or two shades of gray and choose highlight colors contrast well against the background.
- Add highlight colors sparingly—on plots, abort buttons, and perhaps the slider thumbs—for important settings. Small objects need brighter colors and more contrast than larger objects.
- Use differences in contrast more often that differences in color. Color-blind users find it difficult to discern objects when differences are in color rather than contrast.
- Use spacing and alignment to group objects instead of grouping by matching colors.
- Good places to learn about color are stand-alone instrument panels, maps, and magazines.
- Choose objects from the **Controls»All Controls»Dialog Controls** palette if you want your front panel controls to use the system colors.

Spacing and Alignment

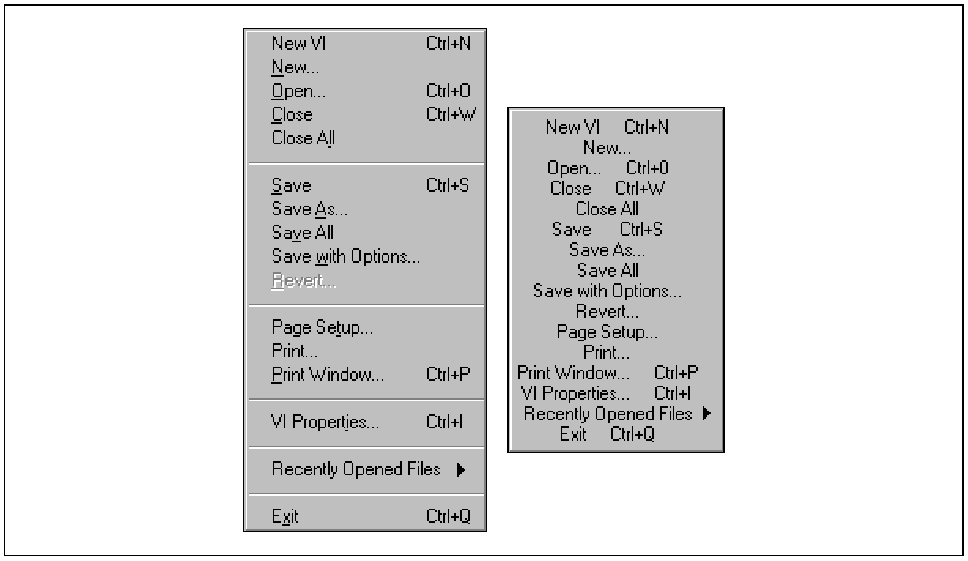
White space and alignment are probably the most important techniques for grouping and separation. The more items that your eye can find on a line, the more cohesive and clean the organization seems. When items are on a line, the eye follows the line from left to right or top to bottom. This is related to the script direction. Although some cultures see items right to left, almost all follow top to bottom.

When you design the front panel, consider how users interact with the VI and group controls and indicators logically. If several controls are related, add a decorative border around them or put them in a cluster.

Centered items are better than random but much less orderly than either left or right alignment. A band of white space acts as a very strong means of alignment. Centered items typically have ragged edges and the order is not as easily noticed.

Do not place front panel objects too closely together. Try to leave some blank space to make the front panel easier to read. Blank space also prevents users from accidentally clicking the wrong control or button.

Menus should be left-justified and related shortcuts should be right-justified as shown in the following example of the LabVIEW **File** menu on the left. It is more difficult to locate items in the center-justified menu as shown in the same example on the right. Notice how the dividing lines between menu sections in the left example help you find the items quickly and strengthen the relationship between the items in the sections.



Avoid placing objects on top of other objects. Placing a label or any other object over or partially covering a control or indicator slows down screen updates and can cause the control or indicator to flicker.

Text and Fonts

Text is easier to read and information is more easily understood when displayed in an orderly way. Use the default LabVIEW fonts. LabVIEW replaces the built-in fonts with comparable font families on different platforms. If you select a different font, LabVIEW substitutes the closest match if the font is unavailable on a computer.

Using too many font styles can make your front panel look busy and disorganized. It is better to use two or three different sizes of the same font. Serifs help people to recognize whole words from a distance. If you are using more than one size of a font, make sure the sizes are noticeably different. If not, it may look like a mistake. Similarly, if you use two different fonts, make sure they are distinct.

Design your front panels with larger fonts and more contrast for industrial operator stations. Glare from lighting, or the need to read information from a distance can make normal fonts difficult to read. Also, remember that touch screens generally require larger fonts and more spacing between selection items.

User Interface Tips and Tools

Some of the built-in LabVIEW tools for making user-friendly front panels include dialog controls, tab controls, decorations, menus, and automatic resizing of front panel objects.

Dialog Controls

A common user interface technique is to display dialog boxes at appropriate times to interact with the user. You can make a VI behave like a dialog box by selecting **File»VI Properties**, selecting the **Window Appearance** category, and selecting the **Dialog** option.

Use the dialog controls located on the **Controls»All Controls»Dialog Controls** palette to create dialog boxes. Because the dialog controls change appearance depending on which platform you run the VI, the appearance of controls in VIs you create will be compatible on all LabVIEW platforms. When you run the VI on a different platform, the dialog controls adapt their color and appearance to match the standard dialog box controls for that platform.

Dialog controls typically ignore all colors except transparent. If you are integrating a graph or non-dialog control into the front panel, try to make them match by hiding some borders or selecting colors similar to the system.

Tab Controls

Physical instruments usually have good user interfaces. Borrow heavily from their design principles but move to smaller or more efficient controls, such as ring controls or tab controls, where appropriate. Use the tab control located on the **Controls»All Controls»Containers** palette to overlap front panel controls and indicators in a smaller area.

To add another page to a tab control, right-click a tab and select **Add Page Before** or **Add Page After** from the shortcut menu. Relabel the tabs with the Labeling tool, and place front panel objects on the appropriate pages. The terminals for these objects are available on the block diagram, as are terminals for any other front panel object.

You can wire the enumerated control terminal of the tab control to the selector of a Case structure to produce cleaner block diagrams. With this method you associate each page of the tab control with a subdiagram, or case, in the Case structure. You place the control and indicator terminals from each page of the tab control—as well as the block diagram nodes and wires associated with those terminals—into the subdiagrams of the Case structure.

Decorations

Use the decorations located on the **Controls»All Controls»Decorations** palette to group or separate objects on a front panel with boxes, lines, or arrows. These objects are for decoration only and do not accept or display data.

Menus

Use custom menus to present front panel functionality in an orderly way and in a relatively small space. Using small amounts of space leaves room on the front panel for critical controls and indicators, items for beginners, items needed for productivity, and items that do not fit well into menus. You also can create keyboard shortcuts for menu items. Refer to Section I, *LabVIEW Run-Time Menus (Optional)*, in this lesson for more information about creating menus.

Automatic Resizing of Front Panel Objects

Use the **VI Properties»Window Size** options to set the minimum size of a window, maintain the window proportion during screen changes, and set front panel objects to resize in two different modes. When you design a VI, consider whether the front panel can display on computers with different screen resolutions. Place a checkmark in the **Maintain proportions of window for different monitor resolutions** checkbox to maintain front panel window proportions relative to the screen resolution.

Most professional applications do not enlarge every control when the window changes size, but you can scale a table, graph, or list with the window, leaving other objects near the window edge. To scale one front panel object with the front panel, select that object and select **Edit»Scale Object with Panel**.

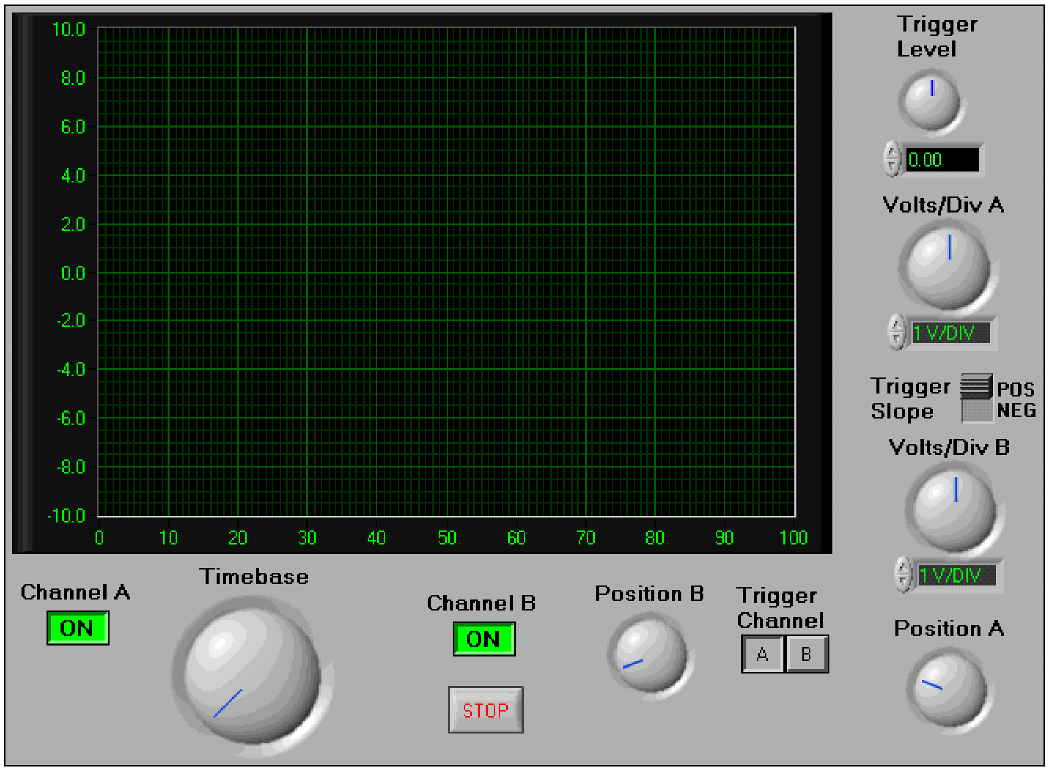
Exercise 2-1 Scope Panel VI

Objective: To logically arrange and separate front panel objects to make the user interface of a VI easier to read and use.

Resize, reorganize, and rearrange the objects on the front panel to make the user interface of a VI easier to use. Set up the graph to resize along with the front panel.

Front Panel

1. Open the Scope Panel VI located in the `C:\Exercises\LabVIEW Basics II` directory.

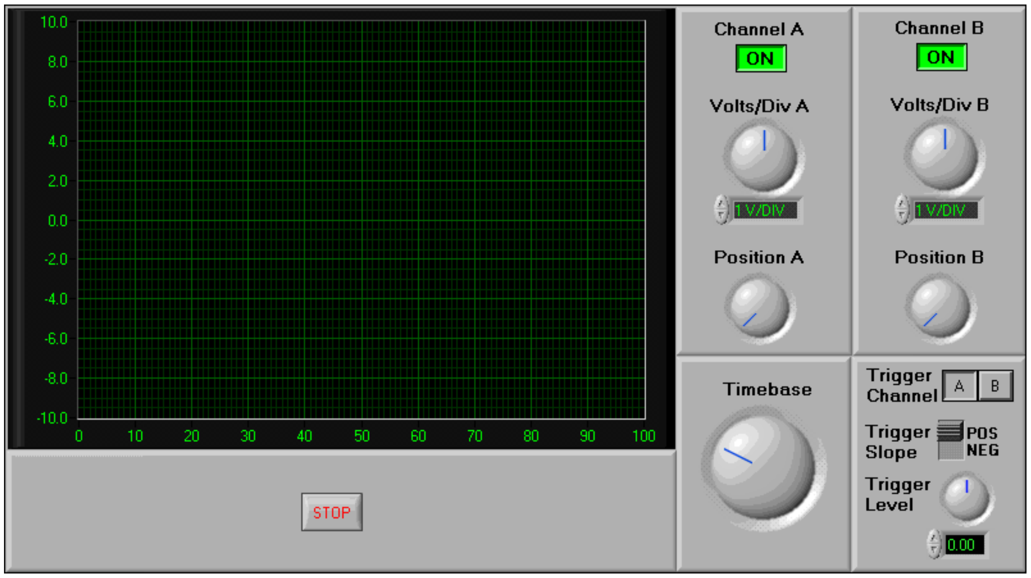


2. Logically group the controls that share similarities. For example, the **Channel A ON/OFF** button, **Position A** knob, and **Volts/Div A** knob all operate on channel A, and it makes sense to have them close to one another. Two other logical groups would be the three Channel B controls and the three trigger controls.

Tip Use the **Align Objects** and **Distribute Objects** features in the toolbar to arrange objects.

After logically grouping the controls, use the Raised Box decoration, located on the **Controls»All Controls»Decorations** palette, to create visible separations between the groups. Click the **Reorder** button on the toolbar and select the **Move to Back** option to move the decorations behind the controls so the controls are visible on top of the raised boxes.

3. Resize your window so the front panel fits inside the window, as shown in the following front panel.



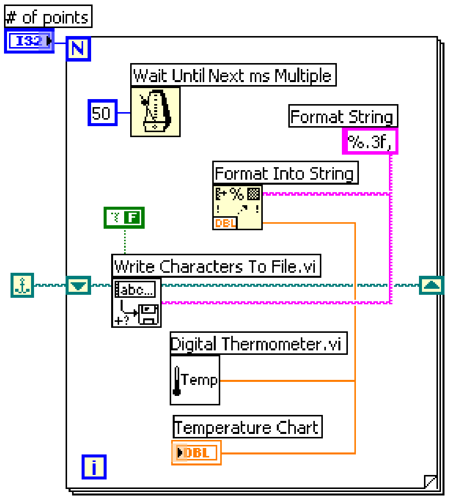
4. Select **File»VI Properties** to display the **VI Properties** dialog box. Select **Window Size** from the top pull-down menu of the **VI Properties** dialog box. In the **Minimum Panel Size** section, click the **Set to Current Window Size** button to set your minimum screen size to the current size of the window. Click the **OK** button to return to the front panel.
5. Select the graph on the front panel, then select **Edit»Scale Object With Panel**. LabVIEW resizes the graph when the entire window is resized and moves the other objects.
6. Save the VI.
7. Resize the front panel window. Notice that the graph resizes with the window and the controls maintain a proportional distance to the graph and each other but do not grow or shrink. However, the decorations do not resize. In order to resize decorations, you must use Boolean buttons that look the same in each state and resize them programmatically using Property Nodes. Refer to Lesson 3, *Object Properties*, for more information about Property Nodes.
8. Close the VI when you are finished.

End of Exercise 2-1

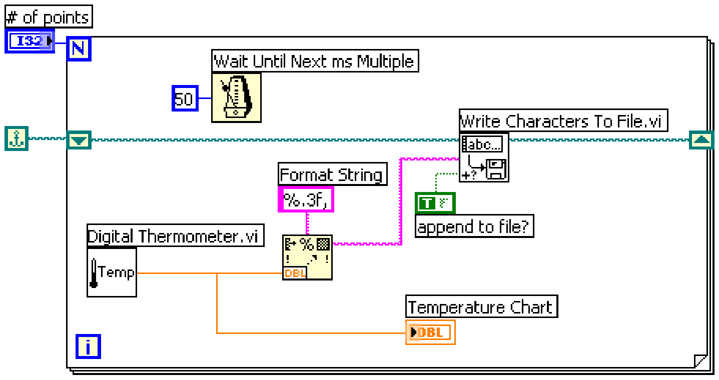
B. Block Diagram Layout

People generally perceive the flow of ideas in a left-to-right and top-to-bottom manner. LabVIEW block diagrams are easiest to follow when they use this same convention. As you learned in the *LabVIEW Basics I: Introduction* course, data flow determines the execution order, so the left to right convention does not affect the operation of the VI. It does, however, make the logic of your block diagrams easier to follow.

For example, the VI in the following example is difficult to understand because it does not conform to the left-to-right and top-to-bottom convention.



The following block diagram uses a more left-to-right and top-to-bottom layout, so it is much easier to read.



C. LabVIEW Wiring Techniques

The block diagram style of programming in LabVIEW allows others to more easily follow the logic of your programs. The techniques you use to wire VIs also can promote understanding. In addition, where you choose to position and place objects on the block diagram is important. The **Align Objects** and **Distribute Objects** tools in LabVIEW allow you to quickly arrange objects on the block diagram to make it easier to see and understand groupings of objects. Placing objects using symmetry and straight lines makes your block diagram easier to read as well as giving it a more professional look.

Keep in mind the following good wiring tips:

- Avoid placing any wires under block diagram objects, such as subVIs or structures.
- Add as few bends in the wires as possible, and try to keep the wires short. Avoid creating wires with long complicated paths that can be confusing.
- Delete any extraneous wires to keep the block diagram clean.
- Avoid the use of local variables when it is possible to pass the data by wire. Every local variable that reads the data makes a copy of the data.
- Try not to pass wires though structures if the data in the wire itself are not used within the structure.
- Evenly space parallel wires in straight lines and around corners.
- Wire directly to tunnels and Shift Registers. Do not run wires underneath structure borders.
- Wire directly to terminals. That is, the terminal that a wire is connected to should be obvious when viewing the block diagram.

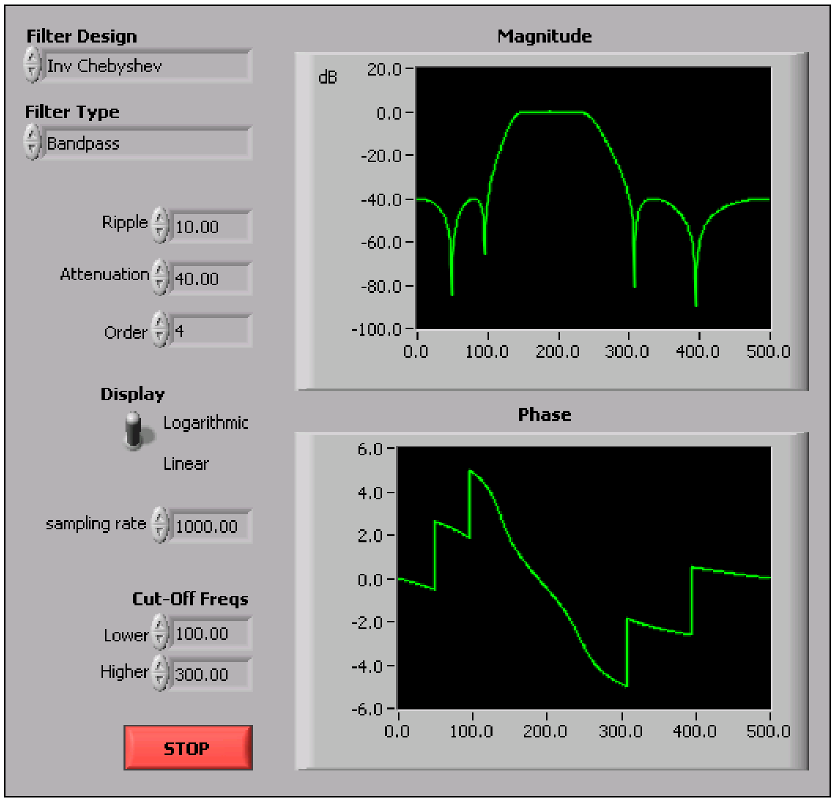
Exercise 2-2 Bad Wiring VI

Objective: To clean up a poorly wired block diagram.

The Bad Wiring VI illustrates how a poorly wired block diagram is difficult to read. Use the wiring techniques described in this lesson to clean up the block diagram.

Front Panel

1. Open the Bad Wiring VI located in the `C:\Exercises\LabVIEW Basics II` directory.

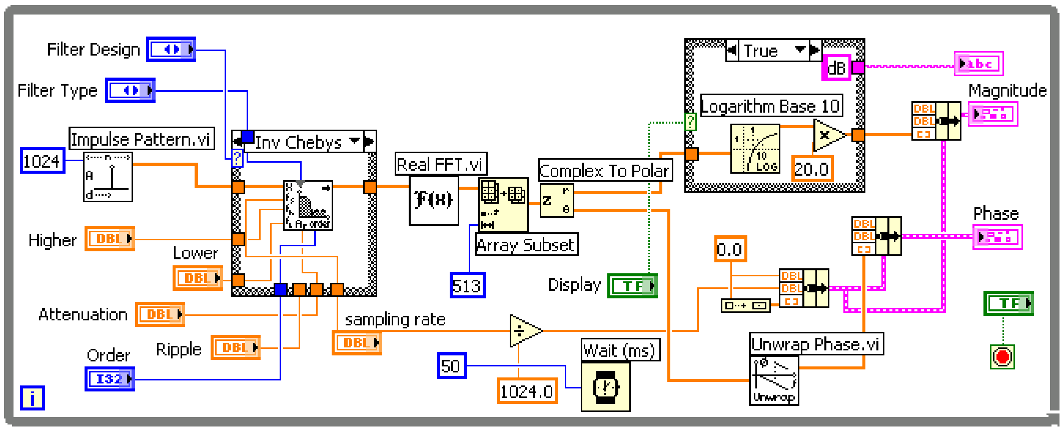


Filter Design: Inv Chebyshev
Filter Type: Bandpass
Ripple: 10.00
Attenuation: 40.00
Order: 4
Display: Logarithmic / Linear
sampling rate: 1000.00
Cut-Off Freqs
Lower: 100.00
Higher: 300.00

2. Run the VI. This VI performs basic filtering and displays the phase of the waveform.

Block Diagram

3. Stop the VI and open the block diagram.



4. Clean up the block diagram to make it easier to read. Use the following tips as a guide:
   - Use left-to-right and top-to-bottom layout.
   - Use the **Align Objects** and **Distribute Objects** pull-down menus to arrange terminals, subVIs, and other block diagram elements.
   - Arrange terminals in a way that corresponds to the terminals they connect to on subVIs.
   - Place terminal labels in different locations to maximize room. For example, you can place the labels immediately to the left of terminals to make it easier to arrange them into vertical columns.
   - Minimize bends in wires.
   - Double- and triple-click poorly arranged wires to find the corresponding terminals.
   - Add extra space to structures and between block diagram elements to ease clutter.
   - Automatically route an existing wire by right-clicking the wire and selecting **Clean Up Wire** from the shortcut menu.

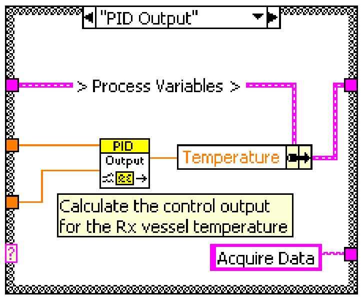
Tip If you make a mistake, use the **Edit»Undo** menu item to return to an earlier state. As you make changes, save frequently in the event you need to use the **File»Revert** menu item.

5. (Optional) Add block diagram comments to further clarify the purpose of different block diagram elements.
6. Save your VI as `Good Wiring.vi` in the `C:\Exercises\LabVIEW Basics II` directory. For an example of how you can clean up this VI, open the Bad Wiring (solution) VI located in the `C:\Solutions\LabVIEW Basics II` directory.

End of Exercise 2-2

D. Commenting in Block Diagrams

Professional developers who maintain and modify VIs know the value of good documentation. While the graphical nature of LabVIEW aids in self-documentation of block diagrams, extra comments are helpful when modifying your VIs in the future. There are two types of block diagram comments—comments that describe the function or operation of algorithms and comments that explain the purpose of data that passes through wires. Both types of comments are shown in the following block diagram. You can insert standard labels either with the Labeling tool, or by inserting a free label from the **Functions»All Functions»Decorations** subpalette. Free labels have a yellow background color.



Use the following guidelines for commenting your VIs:

- Use comments on the block diagrams to explain what the code is doing. Free Labels with yellow backgrounds are standard for block diagram comments.
- Omit labels on obvious functions.
- Label wires to show the function of the data they carry. This is particularly useful for wires coming from Shift Registers.
- Label structures to specify the main functionality of the structure.
- Label constants to specify the nature of the constant.
- Use comments to document algorithms that you use on block diagrams. If you use an algorithm from a book or other reference, provide the reference information.

E. Block Diagram Checklist

Use the following checklist to ensure you use proper block diagram design in your VIs.

- ❑ Avoid creating extremely large block diagrams. Limit the scrolling necessary to see the entire block diagram to one direction.
- ❑ Label controls, important functions, constants, property nodes, local variables, global variables, and structures appropriately.
- ❑ Add comments. Use object labels instead of free labels where applicable and scrollable string constants for long comments.
- ❑ Use standard size fonts for comments and labels.
- ❑ Right-justify text if you place a label to the left of an object.
- ❑ Use standard, consistent font conventions throughout.
- ❑ Use **Size to Text** for all text and add carriage returns if necessary.
- ❑ Reduce white space in smaller block diagrams but allow at least three or four pixels between objects.
- ❑ Flow data from left to right. When possible, wires should enter from the left and exit to the right, not the top or the bottom.
- ❑ Align and distribute functions, terminals, and constants.
- ❑ Label long wires with small labels with white backgrounds.
- ❑ Do not wire behind objects.
- ❑ Make good use of reusable, testable subVIs.
- ❑ Make sure the program can deal with error conditions and invalid values.
- ❑ Show the name of source code or include source code for any CINs.
- ❑ Save with the most important or the first frame of structures showing.
- ❑ Review for efficiency, especially data copying, and accuracy, especially parts without data dependency.

F. Error Handling

In the *LabVIEW Basics I: Introduction* course, you used the **error in** and **error out** clusters to pass error information between functions and subVIs. VIs and functions return errors in one of two ways—with numeric error codes or with an error cluster. Typically, functions use numeric error codes, and VIs use an error cluster, usually with error inputs and outputs.

Error handling in LabVIEW follows the dataflow model. Just as data flows through a VI, so can error information. Wire the error information from the beginning of the VI to the end. Include an error handler VI at the end of the VI to determine if the VI ran without errors. Use the **error in** and **error out** clusters in each VI you use or build to pass the error information through the VI.

As the VI runs, LabVIEW tests for errors at each execution node. If LabVIEW does not find any errors, the node executes normally. If LabVIEW detects an error, the node passes the error to the next node without executing. The next node does the same thing, and so on. At the end of the execution flow, LabVIEW reports the error.

Automatic Error Handling

By default, LabVIEW automatically handles any error that occurs when a VI runs by suspending execution, highlighting the subVI or function where the error occurred, and displaying an error dialog box.

Error Handler VIs

The Simple Error Handler VI accepts the **error in** cluster or the **error code** value and, if an error occurs, displays a dialog box that describes the error and possible reasons for it. You can change the type of dialog box it opens from displaying an **OK** button to display no dialog box or to display a dialog box that gives the user a choice to continue or stop the VI.

In many cases you may not want a dialog box to display when an error occurs. For example, if an I/O VI on the block diagram times out, you might not want the entire application to stop. You also might want the VI to retry for a certain period of time. In LabVIEW, you can make these error handling decisions on the block diagram of the VI.

The General Error Handler VI also accepts the **error in** cluster or the **error code** value and displays a dialog box of the type specified when an error occurs. In addition, you can set up error exceptions so that the VI clears or sets specified errors when they occur. You can also use the General Error Handler VI to add errors to the internal error description table. The error description table describes all errors for LabVIEW and its associated

I/O operations. Therefore, you can add your own error codes and descriptions to the error handler VIs. Refer to the *LabVIEW Help* for information about creating user-defined errors.

If you have separate lines of operations that run in parallel in LabVIEW and each operation maintains its own error clusters, use the Merge Errors VI to combine several error clusters into one. The Merge Errors VI looks at the incoming error clusters or the array of error clusters and outputs the first error found. If the VI finds no errors, it looks for warnings and returns the first warning found. If the VI finds no warnings, it returns no error.

Using While Loops for Error Handling

You can wire an error cluster to the conditional terminal of a While Loop to stop the iteration of the While Loop. When you wire the error cluster to the conditional terminal, only the TRUE or FALSE value of the **status** parameter of the error cluster is passed to the terminal. When an error occurs, the While Loop stops.

When an error cluster is wired to the conditional terminal, the shortcut menu items **Stop if True** and **Continue if True** change to **Stop on Error** and **Continue while Error**.

Using Case Structures for Error Handling

When you wire an error cluster to the selector terminal of a Case structure, the case selector label displays two cases, `Error` and `No Error`, and the border of the Case structure changes color—red for `Error` and green for `No Error`. If an error occurs, the Case structure executes the `Error` subdiagram. Refer to the *Case and Sequence Structures* section of Chapter 8, *Loops and Structures*, of the *LabVIEW User Manual* for more information about using Case structures.

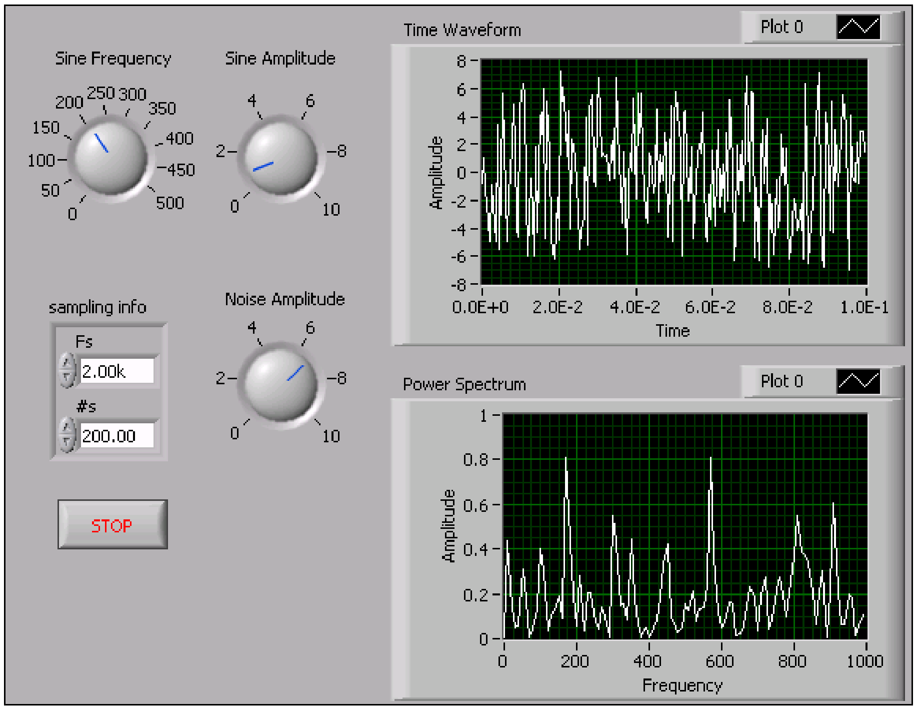
Exercise 2-3 Acquire Data VI

Objective: To build a VI that acquires, analyzes, and presents data while using error handling techniques.

In this exercise, you build a VI that acquires a noisy sine waveform, computes the frequency response of the data, and plots the time and frequency waveforms in waveform graphs. You use the error clusters and the error handling VIs to properly monitor error conditions.

Front Panel

1. Open a new VI and build the following front panel.



   a. Place three Dials, located on the **Controls»Numeric Controls** palette, on the front panel.

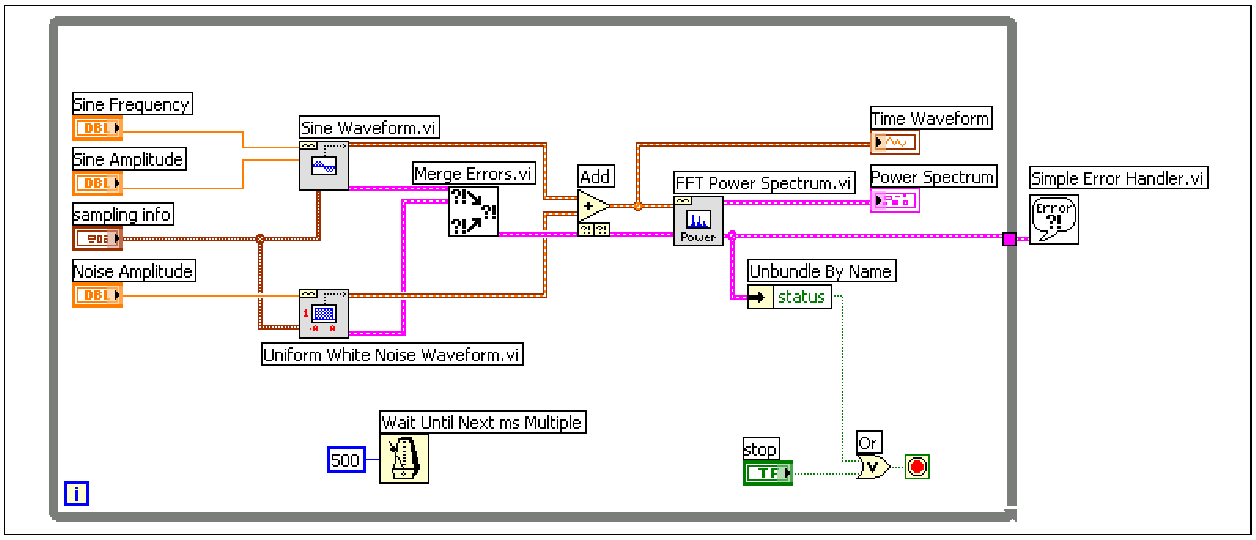
   b. Place a **Stop** button, located on the **Controls»Buttons & Switches** palette, on the front panel.

   c. Place two Waveform Graphs, located on the **Controls»Graph Indicators** palette, on the front panel.

   d. Adjust the range of the Sine Frequency dial to `0.0` to `500.0`. You will create the **sampling info** cluster on the block diagram in the next step. This ensures that the cluster order is correct for the cluster.

Block Diagram

2. Open and build the following block diagram.



a. Place a While Loop, located on the **Functions»Execution Control** palette, on the block diagram. This structures the VI to continue to acquire and analyze data until the user clicks the **Stop** button.

b. Place the Sine Waveform VI, located on the **Functions»All Functions»Analyze»Waveform Generation** palette, on the block diagram. This VI generates a sine waveform with the specified frequency, amplitude, and sampling information. Right-click the **sampling info** input and select **Create»Control** from the shortcut menu. The **sampling info** cluster control appears on the front panel and block diagram.

c. Place the Uniform White Noise Waveform VI, located on the **Functions»All Functions»Analyze»Waveform Generation** palette, on the block diagram. This VI generates a uniform white noise waveform specified by the amplitude and sampling information.

d. Place the Merge Errors VI, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. This VI combines the error clusters coming from the Sine Waveform VI and Uniform White Noise Waveform VI into a single error cluster.

e. Place the Add function, located on the **Functions»Arithmetic & Comparison»Express Numeric** palette, on the block diagram. This function is polymorphic and includes the error input and output terminals when a waveform data type is wired into the terminals.

f. Place the FFT Power Spectrum VI, located on the **Functions»All Functions»Analyze»Waveform Measurements** palette, on the block diagram. This VI calculates the frequency response of the time waveform input and averages the data according to the specified averaging parameters.

g. Place the Wait Until Next ms Multiple function, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. This function causes the While Loop to execute every half second. Right-click the input terminal and select **Create»Constant** from the shortcut menu. Type `500` in the constant.

h. Place the Unbundle By Name function, located on the **Functions» All Functions»Cluster** palette, on the block diagram. This function extracts the **status** Boolean from the error cluster in order to stop the loop if an error occurs.

i. Place the Or function, located on the **Functions»Arithmetic & Comparison»Express Boolean** palette, on the block diagram. This function combines the error **status** Boolean and the **Stop** button on the front panel so that the loop stops if either of these values is True.

j. Place the Simple Error Handler VI, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. If an error occurs in this VI, a dialog box appears and displays the error information.

3. Select **File»Save As** to save the VI as `Acquire Data.vi` in the `C:\Exercises\LabVIEW Basics II` directory. You use this VI later in the course.

4. Observe how the subVIs you used in this block diagram use error handling.

   a. Double-click the Sine Waveform VI and open its block diagram. Notice that it first checks the **error in** cluster for previous errors. If an error has occurred, LabVIEW returns an empty waveform and passes out the error information. If no error has occurred, LabVIEW generates a sine waveform of the specified input parameters.

   b. Close the Sine Waveform VI when you are finished.

5. Display the front panel and run the Acquire Data VI. Adjust the front panel controls to see the time and frequency waveforms change. Force an error by entering incorrect values into the controls. For example, a sampling frequency, F_s, too low or too high results in an error.

6. Stop the VI when you are finished.

End of Exercise 2-3

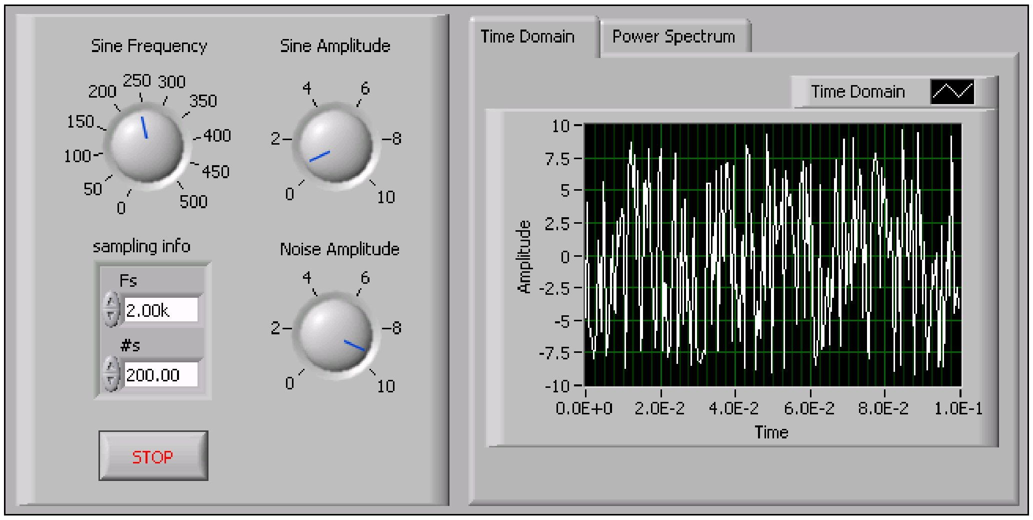
Exercise 2-4 Enhanced Acquire Data VI

Objective: To modify a VI to use the tab control along with proper user-interface design techniques.

In this exercise, you modify the Acquire Data VI you built in Exercise 2-3 so that it uses the tab control and a decoration. One tab displays the time waveform data and the other tab displays the power spectrum data.

Front Panel

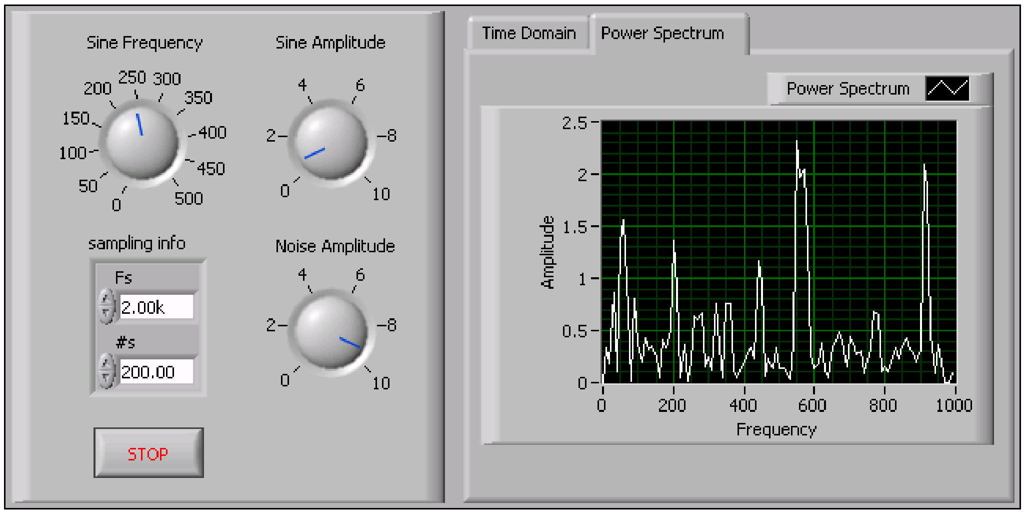
1. Modify the Acquire Data VI.



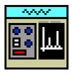
   a. Place a horizontal smooth box decoration, located on the **Controls» All Controls»Decorations** palette, on the front panel. Place the three dials, the sampling info cluster, and the **Stop** button on the decoration.

   b. Place a tab control, located on the **Controls»All Controls» Containers** palette, on the front panel. Select the Time Waveform graph and place it on the first page of the tab control.

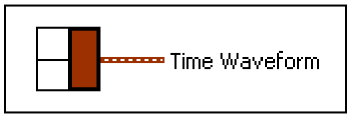
   c. Name the two pages of the tab control `Time Domain` and `Power Spectrum`, respectively. Click the **Power Spectrum** page and add the Power Spectrum graph to the page, as shown on the following front panel.



2. Create an icon for this VI. You will use this VI as a subVI in a later exercise. Right-click the icon in the top right corner of the front panel and select **Edit Icon** from the shortcut menu. Design an icon similar to the following example.



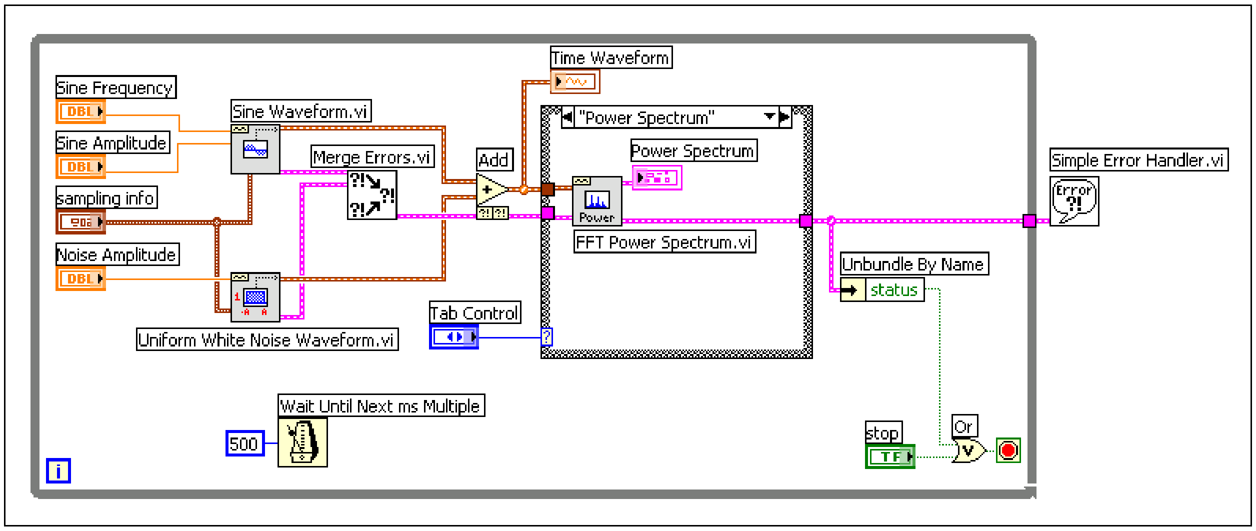
3. Create the connector pane for this VI by right-clicking the icon in the top right corner of the front panel and selecting **Show Connector** from the shortcut menu. Select the pattern and connect the Time Waveform graph to the terminal as shown in the following example.

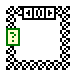


4. Select **File»Save As** to save the VI as `Enhanced Acquire Data.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

Block Diagram

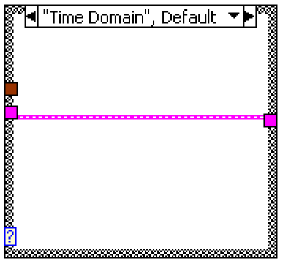
5. Open and modify the block diagram.





a. Place a Case structure, located on the **Functions»Execution Control** palette, on the block diagram. Wire the tab control to the Case structure. One case handles the **Time Domain** page and the other case handles the **Power Spectrum** page. Complete the Power Spectrum case as shown in the previous figure.

b. Complete the Time Domain (Default) case as shown in the following figure.



Using a Case structure in the code increases the performance and usability of the application because the power spectrum processes only when the user needs to see the data.

6. Display the front panel and run the VI. Adjust the front panel controls to change the time and frequency waveforms. Click between the **Time Domain** and **Power Spectrum** pages on the tab control. Stop the VI when you are finished.

7. Make sure each of the inputs has valid non-zero values then select **Operate»Make Current Values Default** to ensure the VI generates valid data when called as a subVI.

8. Save and close the VI when you are finished.

End of Exercise 2-4

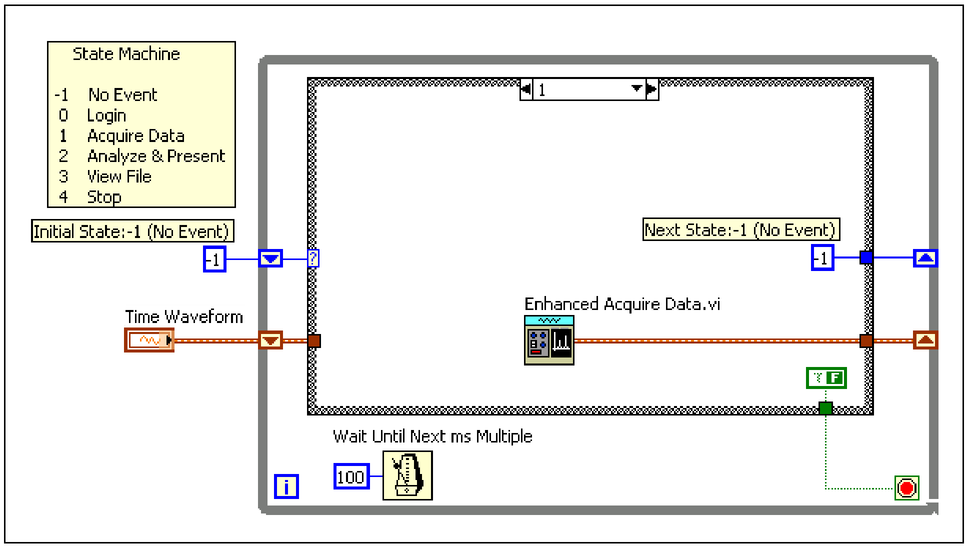
Exercise 2-5 State Machine with Enhanced Acquire Data VI

Objective: To add the Enhanced Acquire Data VI to the application started in Exercise 1-1.

In this exercise, you add the Enhanced Acquire Data VI that you created in Exercise 2-4 to the project.

Block Diagram

1. Open `Menu.vi`, which you created in Exercise 1-1.
2. The front panel is already complete. Open the block diagram.

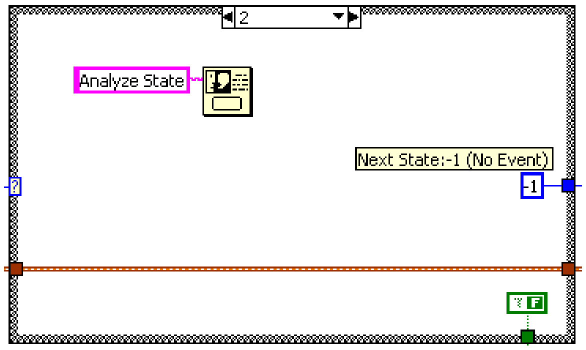


3. Display case 1 of the Case structure and delete the One Button Dialog function. Add the Enhanced Acquire Data VI, which you built in Exercise 2-4.

Note If you did not complete Exercise 2-4, use the Enhanced Acquire Data VI located in the `C:\Solutions\LabVIEW Basics II` directory.

4. Add a Shift Register to the border of the While Loop. Connect the **Time waveform** output of the Enhanced Acquire Data VI to the right side of the Shift Register.
5. Initialize the new Shift Register you created.
   a. Right-click the left side of the Shift Register and select **Create» Control** from the shortcut menu to create an empty **Time Waveform** control on the front panel.
   b. Hide the **Time Waveform** control by right-clicking the terminal and selecting **Hide Control** from the shortcut menu. This makes the waveform control invisible on the front panel so it does not confuse users.

6. Modify the VI Properties of the Enhanced Acquire Data VI so that it appears like a dialog box when it is called.
   a. Double-click the Enhanced Acquire Data VI to open its front panel.
   b. Select **File»VI Properties** and select **Window Appearance** from the top pull-down menu and select the **Dialog** option.
   c. Click the **OK** button, then save and close the Enhanced Acquire Data VI.
7. Remember that if one case in a Case structure passes data out of the case, all other cases in the Case structure must also send out data. Finish wiring the VI so that the data pass through the other cases unchanged. Make sure that the data passes through the other cases correctly, as shown in the following block diagram.



8. Select **File»Save As** to save the VI as `State Machine with Enhanced Acquire Data.vi` in the `C:\Exercises\LabVIEW Basics II` directory.
9. Display the front panel and run the VI to test it.
10. Close the VI when you are finished.

End of Exercise 2-5

G. Hierarchical File Organization

Organize the VIs in the file system to reflect the hierarchical nature of your application. Make the top-level VIs directly accessible. Place subVIs in subdirectories and group them to reflect any modular components you have designed, such as instrument drivers, configuration utilities, and file I/O drivers.

Create a directory for all the VIs that are used by one application and give it a meaningful name, as shown in Figure 2-1. Save the main VIs in this directory and the subVIs in a subdirectory. If the subVIs have subVIs, continue the directory hierarchy downward.

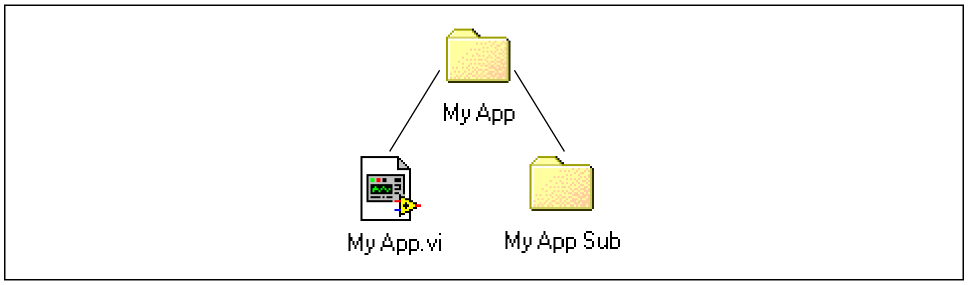


Figure 2-1. Directory Hierarchy

When naming VIs, VI libraries, and directories, avoid using characters that are not accepted by all file systems, such as slash (`/`), backslash (`\`), colon (`:`), tilde (~), and so on. Most operating systems accept long descriptive names for files, up to 31 characters on a Macintosh and 255 characters on other platforms.

Select **Tools»Options**, select **Paths** from the top pull-down menu, then select **VI Search Path** from the second pull-down menu to make sure the VI Search Path contains `<topvi>\*` and `<foundvi>\*`. The `*` causes all subdirectories to be searched. In Figure 2-1, `MyApp.vi` is the top-level VI. This means that the application searches for subVIs in the directory `MyApp`. Once a subVI is found in a directory, the application looks in that directory for subsequent subVIs.

Avoid creating files with the same name anywhere within the hierarchy. Only one VI of a given name can be in memory at a time. If you have a VI with a specific name in memory and you attempt to load another VI that references a subVI of the same name, the VI links to the VI in memory. If you make backup copies of files, be sure to save them into a directory outside the normal search hierarchy so that LabVIEW does not mistakenly load them into memory when you open development VIs.

Refer to Chapter 7, *Creating VIs and SubVIs*, of the *LabVIEW User Manual* for more information about saving VIs individually and in VI libraries.

H. Sequence Structures

The Stacked Sequence and Flat Sequence structures, shown at left, contain multiple subdiagrams, or frames, which execute in sequential order. A sequence structure executes frame 0, then frame 1, then frame 2, until the last frame executes. The Stacked Sequence structure does not complete execution or return any data until the last frame executes. Frames in a Flat Sequence structure execute in order and when all data wired to the frame is available. The data leave each frame as the frame finishes executing. Use the Stacked Sequence structure if you want to conserve space on the block diagram. Use the Flat Sequence structure to avoid using sequence locals and to better document the block diagram.

◀ 0 [0..2] ▶

The structure selector label at the top of the Stacked Sequence structure, shown at left, contains the current frame number and range of frames in the center and decrement and increment arrow buttons on each side. For example, in the sequence selector label shown at left, `0` is the current frame number and `[0..2]` is the range of frames. Click the decrement and increment arrow buttons to scroll through the available frames.

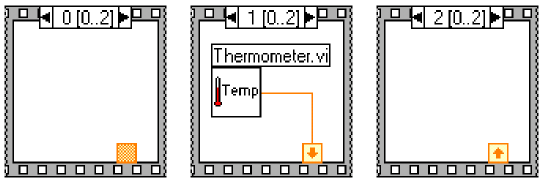
Use the sequence structures to control the execution order when natural data dependency does not exist. A node that receives data from another node depends on the other node for data and always executes after the other node completes execution. Within each frame of a sequence structure, as in the rest of the block diagram, data dependency determines the execution order of nodes.

The tunnels of Stacked Sequence structures can have only one data source, unlike Case structures. The output can emit from any frame, but data leave the Stacked Sequence structure only when all frames complete execution, not when the individual frames complete execution. As with Case structures, data at input tunnels are available to all frames.

Sequence Locals

To pass data from one frame to any subsequent frame of a Stacked Sequence structure, use a sequence local terminal, shown at left. An outward-pointing arrow appears in the sequence local terminal of the frame that contains the data source. The terminal in subsequent frames contains an inward-pointing arrow, indicating that the terminal is a data source for that frame. You cannot use the sequence local terminal in frames that precede the first frame where you wired the sequence local. Right-click the Stacked Sequence structure border and select **Add Sequence Local** from the shortcut menu to create a sequence local.

The following example shows three frames of a Stacked Sequence structure. A sequence local in frame 1 takes the value that the Thermometer VI returns and passes it to frame 2, as indicated by the arrow pointing into frame 2. This value is not available in frame 0, as indicated by the dimmed square.

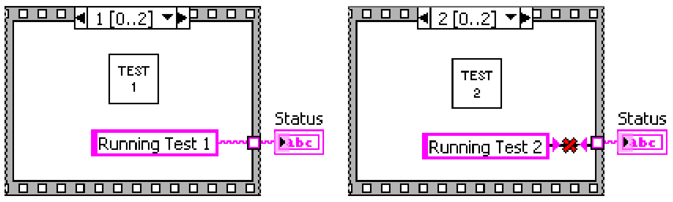


Avoid Overusing Sequence Structures

To take advantage of the inherent parallelism in LabVIEW, avoid overusing sequence structures. Sequence structures guarantee the order of execution and prohibit parallel operations. For example, asynchronous tasks that use I/O devices, such as PXI, GPIB, serial ports, and DAQ devices, can run concurrently with other operations if sequence structures do not prevent them from doing so. Stacked Sequence structures also hide sections of the block diagram and interrupt the natural left-to-right flow of data.

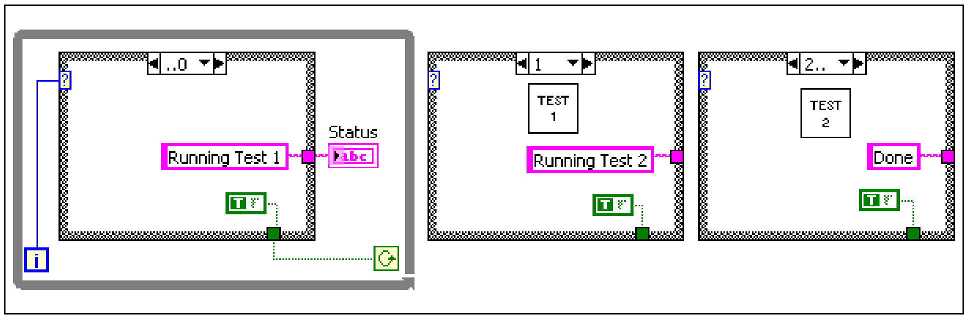
When you need to control the execution order, consider establishing data dependency between the nodes. For example, you can use error I/O to control the execution order of I/O. Refer to the *Error Checking and Error Handling* section of Chapter 6, *Running and Debugging VIs*, of the *LabVIEW User Manual* for more information about error I/O.

Also, do not use sequence structures to update an indicator from multiple frames of the sequence structure. For example, a VI used in a test application might have a **Status** indicator that displays the name of the current test in progress. If each test is a subVI called from a different frame, you cannot update the indicator from each frame, as shown by the broken wire in the following block diagram.



Because all frames of a sequence structure execute before any data pass out of the structure, only one frame can assign a value to the **Status** indicator.

Instead, use a Case structure and a While Loop, as shown in the following block diagram.



Each case in the Case structure is equivalent to a sequence structure frame. Each iteration of the While Loop executes the next case. The **Status** indicator displays the status of the VI for each case. The **Status** indicator is updated in the case prior to the one that calls the corresponding subVI because data pass out of the structure after each case executes.

Unlike a sequence structure, a Case structure can pass data to end the While Loop during any case. For example, if an error occurs while running the first test, the Case structure can pass FALSE to the conditional terminal to end the loop. However, a sequence structure must execute all its frames, even if an error occurs.

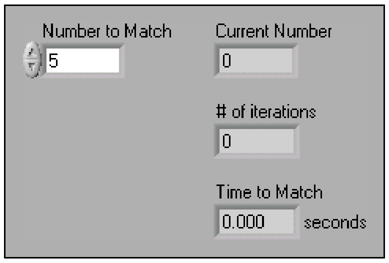
Exercise 2-6 Time to Match VI

Objective: To use the Stacked Sequence structure.

Complete the following steps to build a VI that computes the time it takes to generate a random number that matches a number you specify. This exercise shows a good use of a Stacked Sequence structure.

Front Panel

1. Open the Auto Match VI located in the `C:\Exercises\LabVIEW Basics II` directory.
2. Enter a number in **Number to Match** and run the VI. Notice that the VI continues to run until the random number generated matches the number you specified.
3. Modify the front panel as follows.

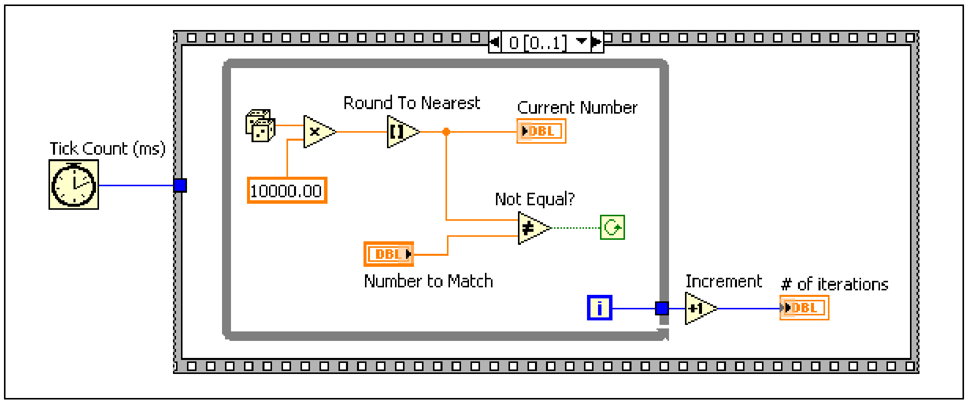


   a. Add a Numeric indicator to the front panel.

   b. Change the label of the indicator to **Time to Match**.
4. Select **File»Save As** to save the VI as `Time to Match.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

Block Diagram

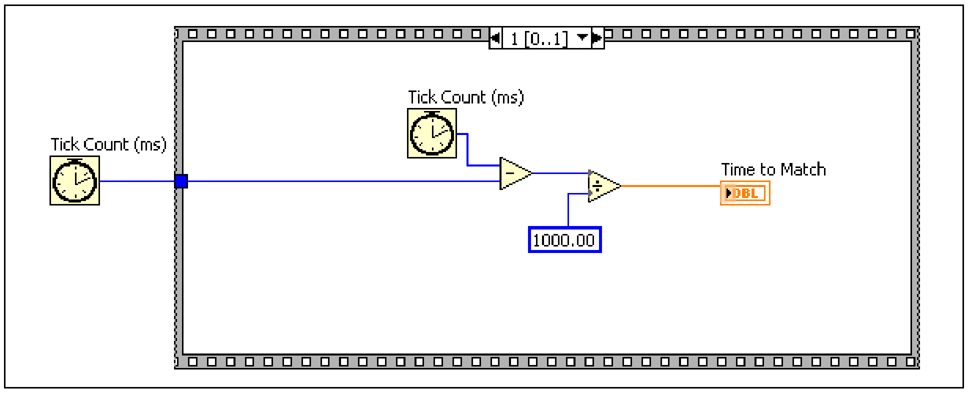
5. Modify the block diagram.



a. Place a Stacked Sequence structure, located on the **Functions»All Functions»Structures** palette, on the block diagram. Right-click the structure border and select **Add Frame After** from the shortcut menu to add a frame. Make sure the While Loop is contained in frame 0 of the Stacked Sequence structure.

b. Place the Tick Count (ms) function, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. This function reads the current value of the operating system clock and returns the value in milliseconds.

c. Display frame 1 of the Stacked Sequence structure.



d. Place the Tick Count (ms) function, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram.

e. Move the Time to Match indicator into frame 1 of the Stacked Sequence structure.

f. Place the Subtract and Divide functions, located on the **Functions» Arithmetic & Comparison»Express Numeric** palette, on the block diagram.

6. Save the VI.
7. Display the front panel, enter a number in **Number to Match**, and run the VI.

   In frame 0, the VI executes the While Loop while **Current Number** does not match **Number to Match**. In frame 1, the Tick Count (ms) function reads the operating system clock. The VI subtracts the new value from the initial time read and returns the elapsed time in seconds.

Note If **Time to Match** is always `0.000`, the VI might be running too quickly. Either run the VI with execution highlighting enabled or increase the numeric constant wired to the Multiply function in frame 0 to a large value, such as `1,000,000`.

8. Close the VI.

End of Exercise 2-6

I. LabVIEW Run-Time Menus (Optional)

You can create custom menus for every VI you build using the **Menu Editor** dialog box, and you can configure VIs to show or hide menu bars.

Note Custom menus appear only while the VI runs.

You can build custom menus or modify the default LabVIEW menus statically when you edit the VI or programmatically when you run the VI.

Static Menus

When you select **Edit»Run-Time Menu** and create a menu in the **Menu Editor** dialog box, LabVIEW creates a run-time menu (`.rtm`) file so you can have a custom menu bar on a VI rather than the default menu bar. After you create and save the `.rtm` file, you must maintain the same relative path between the VI and the `.rtm` file. Use the **Menu Editor** dialog box to associate a custom `.rtm` file with a VI. When the VI runs, it loads the menu from the `.rtm` file.

Menu items can be the following three types:

- **User Item**—Allows you to enter new items that must be handled programmatically on the block diagram. A user item has a name, which is the string that appears on the menu, and a tag, which is a unique, case-insensitive string identifier. The tag identifies the user item on the block diagram. When you type a name, LabVIEW copies it to the tag. You can edit the tag to be different from the name. For a menu item to be valid, its tag must have a value. The **Item Tag** text box displays question marks for invalid menu items. LabVIEW ensures that the tag is unique to a menu hierarchy and appends numbers when necessary.
- **Separator**—Inserts a separation line on the menu. You cannot set any attributes for this item.
- **Application Item**—Allows you to select default menu items. To insert a menu item, select **Application Item** and follow the hierarchy to the items you want to add. Add individual items or entire submenus. LabVIEW handles application items automatically. These item tags do not appear in block diagrams. You cannot alter the name, tag, or other properties of an application item. LabVIEW reserves the prefix `APP_` for application item tags.

Click the blue + button, shown at left, in the toolbar to add more items to the custom menu. Click the red `X` button, shown at left, to delete items. You can arrange the menu hierarchy by clicking the arrow buttons in the toolbar, using the hierarchy manipulation options in the **Edit** menu, or by dragging and dropping.

Menu Selection Handling

Use the functions located on the top row of the **Functions»All Functions» Application Control»Menu** palette to handle menu selections.

When you create a custom menu, you assign each menu item a unique, case-insensitive string identifier called a tag. When the user selects a menu item, you retrieve its tag programmatically using the Get Menu Selection function. LabVIEW provides a handler on the block diagram for each menu item based on the tag value of each menu item. The handler is a While Loop and Case structure combination that allows you to determine which, if any, menu is selected and to execute the appropriate code.

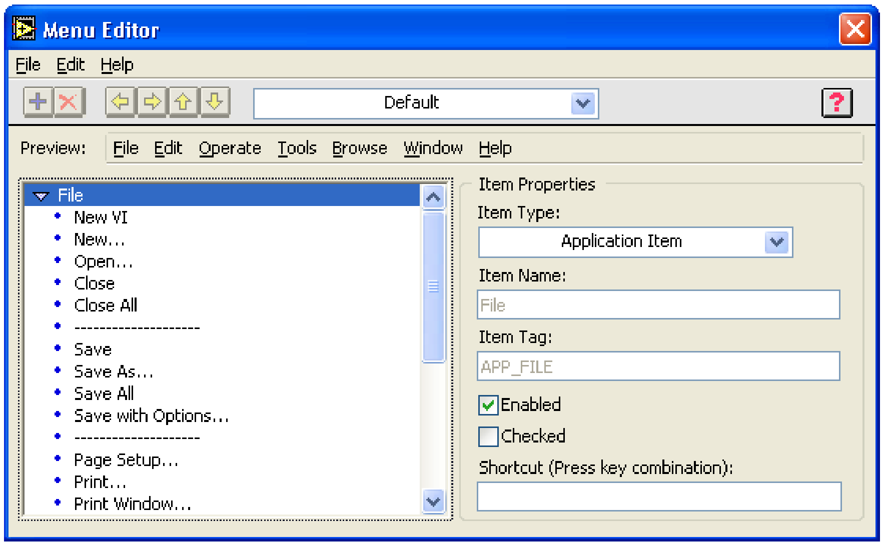
After you build a custom menu, build a Case structure on the block diagram that executes, or handles, each item in the custom menu. This process is called menu selection handling. Use the Get Menu Selection and Enable Menu Tracking functions to define which actions to take when users select each menu item. LabVIEW automatically handles all application items.

Exercise 2-7 Pull-down Menu VI (Optional)

Objective: To build a VI using a custom run-time menu.

This VI illustrates how to edit and programmatically control a custom menu in a LabVIEW application. Build a custom run-time menu for this VI and modify the block diagram so that you can access the custom menu.

1. Open the Pull-down Menu VI located in the `C:\Exercises\LabVIEW Basics II` directory. The front panel and block diagram are partially complete.
2. Select **Edit»Run-Time Menu** to display the **Menu Editor** dialog box.



The current run-time menu for the application is the LabVIEW default menu. In the next several steps, you replace that menu with a custom list of selections.

a. Change the menu type from **Default** to **Custom** in the top pull-down menu of the **Menu Editor** dialog box. The menu listed on the left portion of the dialog box should be replaced with a line of question marks, representing a single unnamed item.

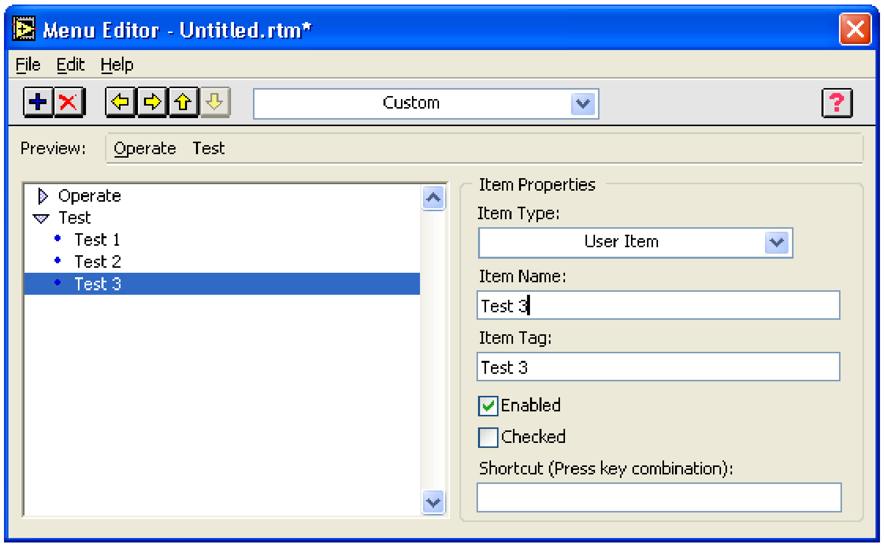
b. In the **Item Type** pull-down menu, select **Application Item»Operate»Entire Menu**. The LabVIEW **Operate** menu should be added to the custom menu.

c. Take a moment to navigate the **Operate** menu in the editor. Notice that you can select items and collapse submenus using the triangle icons. As you select individual items in the menu, the corresponding **Item Name** and **Item Tag** appear in the **Item Properties** section of the editor. When finished, collapse the **Operate** menu by clicking the triangle next to the **Operate** option. You should now see only the **Operate** item in the menu list.

d. Click the **+** button in the **Menu Editor** toolbar. A new unnamed item, `???`, appears in the menu list. With this item highlighted, type `Test` in the **Item Name** property. This menu item now has an item name and tag of **Test**.

e. Click the **+** button again to add another entry under the **Test** item. Click the right arrow button on the toolbar, and this unnamed option becomes a subitem under the **Test** menu. Type in the Item Name `Test 1` for this new item.

f. Add two more subitems under the **Test** submenu called `Test 2` and `Test 3`. The **Menu Editor** dialog box should now resemble the following example. The **Preview** section of the dialog box shows how the custom menu will behave during run time.



Custom

Preview: Operate Test

- Operate
- Test
  - Test 1
  - Test 2
  - Test 3

Item Type: User Item

Item Name: Test 3

Item Tag: Test 3

✓ Enabled

☐ Checked

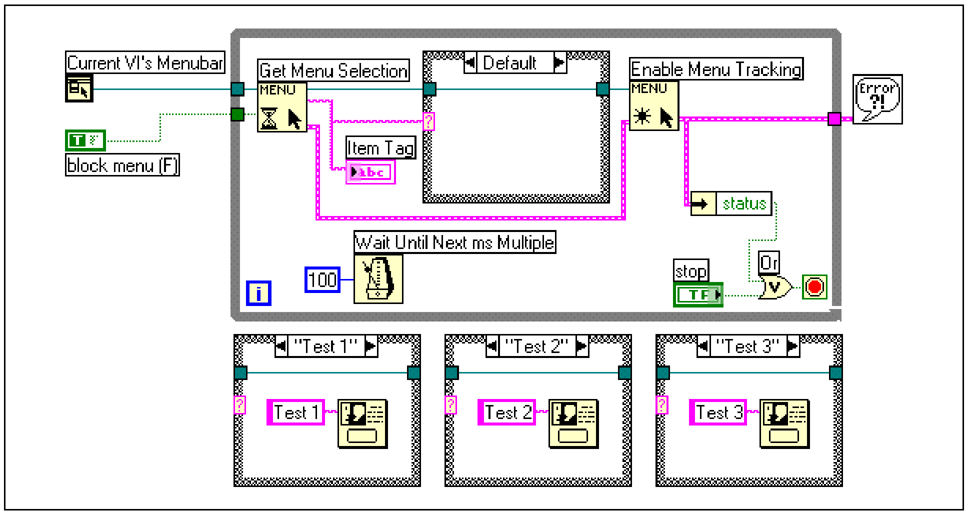
Shortcut (Press key combination):

g. Select **File»Save** from the **Menu Editor** dialog box. Save the run-time menu as `Menu Exercise.rtm` in the `C:\Exercises\LabVIEW Basics II` directory.

h. Close the **Menu Editor** dialog box. When LabVIEW asks if you want to change the run-time menu to `Menu Exercise.rtm`, select **Yes**. You have configured a custom menu that appears while the VI executes.

Block Diagram

3. Open the block diagram of the VI and complete it as shown in the following example.



a. Place the Current VI's Menubar function, located on the **Functions»All Functions»Application Control»Menu** palette, on the block diagram. This function returns the refnum for the selected VI's menu so that it can be manipulated.

b. Place the Get Menu Selection function, located on the **Functions»All Functions»Application Control»Menu** palette, on the block diagram. Each time the While Loop executes, the Get Menu Selection function returns the **Item Tag** for any user item selected in the run-time menu. If no user item is selected, **Item Tag** returns an empty string. This function is configured so that every time it reads the menu bar, it prevents the user from making another menu selection until Enable Menu Tracking executes.

c. Place the Enable Menu Tracking function, located on the **Functions»All Functions»Application Control»Menu** palette, on the block diagram. This function enables the run-time menu, after it has been disabled by the Get Menu Selection function.

d. Place the Unbundle By Name function, located on the **Functions»All Functions»Cluster** palette, on the block diagram. This function extracts the error **status** value.

e. Place the Or function, located on the **Functions»Arithmetic & Comparison»Express Boolean** palette, on the block diagram. This function causes the loop to stop if the user clicks the **STOP** button or an error occurs.

4. Save the VI.

5. Display the front panel and run the VI. When the VI executes, the custom run-time menu appears on the front panel. If you select one of the items in the **Test** menu, that item's name appears in the **Item Tag** indicator and a dialog box appears with the test name in it. At this point, if you try to select another menu item, the menu is disabled by the **block menu** parameter of the Get Menu Selection function. When you click the **OK** button on the dialog box, the **Item Tag** indicator is cleared and the menu is re-enabled by the Enable Menu Tracking function. Also, notice that the items from the **Operate** menu do not appear in the **Item Tag** string—only user items are returned from the Get Menu Selection option.

   To observe the flow of the VI, you might want to turn on Execution Highlighting and Single-Stepping and examine the block diagram. Click the **Stop** button on the front panel to halt program execution.

6. Select **Edit»Run-Time Menu** and use the **Menu Editor** to assign shortcut keys to the user items you created. Assign the following keyboard shortcuts to the test options.

Menu Item	Windows Keyboard Shortcut	Macintosh Keyboard Shortcut
Test 1	<Ctrl-1>	<Option-1>
Test 2	<Ctrl-2>	<Option-2>
Test 3	<Ctrl-3>	<Option-3>

7. On a Windows platform you also can assign an <Alt> keyboard shortcut to menu items. This is accomplished by preceding the letter of the menu name for the shortcut with an underscore. For example, to assign <Alt-X> to a menu item called Execute, give the **Execute** option an Item Name of `E_xecute`. In this exercise, assign <Alt-T> to the **Test** option in the menu.
8. Save the menu with the updated keyboard shortcuts and run the VI. Now you should be able to use the keyboard shortcuts, rather than the cursor, to select the different test options.
9. Close the VI.

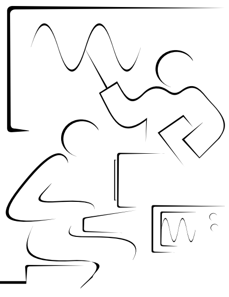
End of Exercise 2-7

Summary, Tips, and Tricks

- When you are designing user interfaces, keep the following things in mind: the number of objects in the panel, color, spacing and alignment of objects, and the text and fonts used.
- LabVIEW contains the following tools to help you create user interfaces: dialog controls, tab controls, decorations, menus, and automatic resizing of panel objects.
- When designing block diagrams, keep the following things in mind: hierarchical design, data flow, good wiring techniques, and comments.
- Error clusters are a powerful method of error handling used with nearly all of the I/O VIs and functions in LabVIEW. These clusters pass error information from one VI to the next.
- An error handler at the end of the data flow can receive the error cluster and display error information in a dialog box.
- To create your own run-time menus, select **Edit»Run-Time Menu** to display the **Menu Editor** dialog box.
- You can programmatically create and read run-time menus with the functions in the **Functions»All Functions»Application Control» Menu** palette.

Notes

Lesson 3
Object Properties



This lesson describes the use of object properties to modify front panel objects. Properties you can modify include color, size, position, behavior, and more. The first part of the lesson focuses on properties for simple front panel objects. The second part of the lesson describes properties associated with graphs and charts. Finally, this lesson describes control references and their advantages in modifying properties from subVIs.

You Will Learn:

A. About Property Nodes

B. About Graph and Chart Properties

C. How to use Control References

A. Property Nodes

In some applications, you might want to programmatically modify the appearance of front panel objects in response to certain inputs. For example, if a user enters an invalid password, you might want a red LED to start blinking. Another example would be changing the color of a trace on a chart. When data points are above a certain value, you might want to show a red trace instead of a green one. Property Nodes allow you to make these modifications programmatically. You can also use Property Nodes to resize front panel objects, hide parts of the front panel, add cursors to graphs programmatically, and so on.

Property Nodes in LabVIEW are very powerful and have many uses. This lesson describes how to use Property Nodes to change the appearance and function of front panel objects programmatically. Refer to the *LabVIEW User Manual* for more information about Property Nodes.

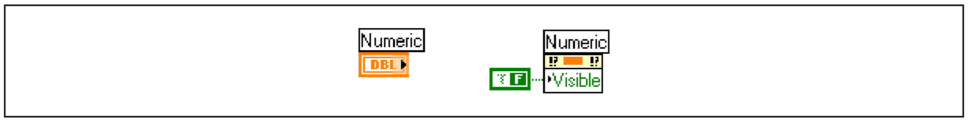
Creating Property Nodes

Create a Property Node by right-clicking an object and selecting **Create» Property Node** from the shortcut menu. LabVIEW creates a Property Node on the block diagram that is implicitly linked to the front panel object. If the object has an owned label, the Property Node has the same label. You can change the label after creating the node. You can also create multiple Property Nodes for the same object.

Using Property Nodes

When you create a Property Node, it initially has one terminal representing a property you can modify for the corresponding front panel object. Using this terminal on the Property Node, you can either set (write) the property or get (read) the current state of that property.

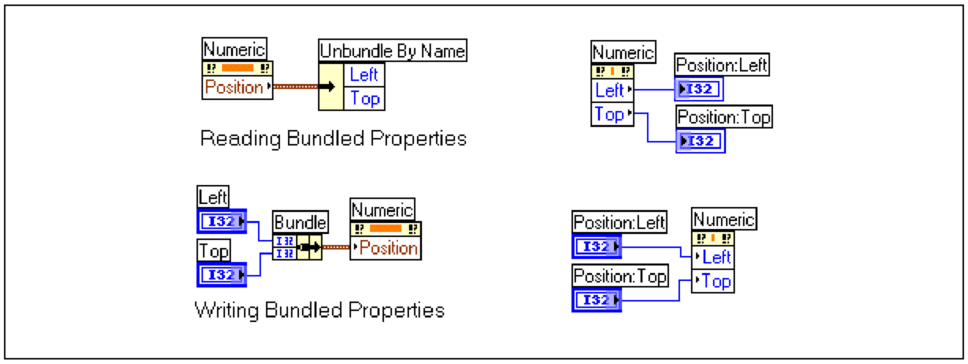
For example, if you create a Property Node for a digital Numeric control, it appears on the block diagram with the Visible property selected by default. A small arrow appears on the right side of that terminal, indicating that you are reading that property value. You can change the action to write by right-clicking the terminal and selecting **Change To Write** from the shortcut menu. Wiring a Boolean FALSE to that property terminal causes the numeric control to vanish from the front panel when the Property Node receives the data. Wiring a Boolean TRUE causes the control to reappear.



To get property information, right-click the node and select **Change to Read** from the shortcut menu. To set property information, right-click the node and select **Change to Write** from the shortcut menu. If the small direction arrow on the property is on the right, you are getting the property value. If the small direction arrow on a property is on the left, you are setting the property value. If the Property Node is set to Read, when it executes it outputs a Boolean TRUE if the control is visible or a Boolean FALSE if it is invisible.

To add terminals to the node, right-click and select **Add Element** from the shortcut menu or use the Positioning tool to resize the node. You then can associate each Property Node terminal with a different property from its shortcut menu.

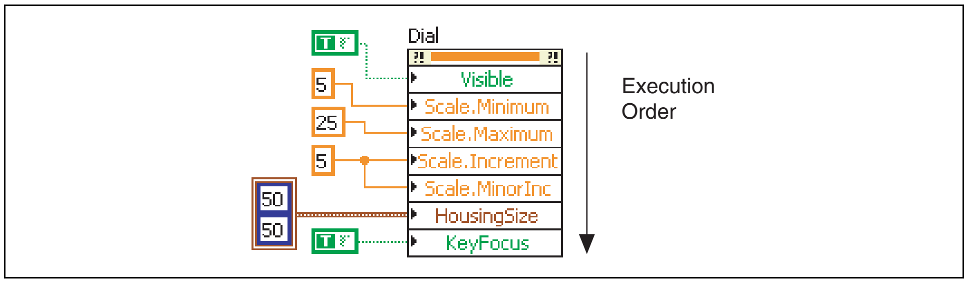
Some properties use clusters. These clusters contain several properties that you can unbundle using the Unbundle function located on the **Functions»All Functions»Cluster** palette. Writing to these properties requires the Bundle function, as shown in the following block diagram.



To access bundled properties, select **All Elements** from the shortcut menu. For example, you can access all the elements in the Position property by selecting **Properties»Position»All Elements** from the shortcut menu.

Property Node Execution Order

Property Nodes execute each terminal in order from top to bottom. For example, in the Numeric Dial Property Node shown in the following example, the control is first made visible. Then the scale values are set, the size of the dial housing is set, and finally the control is given the Key Focus. If an error occurs on a terminal, the node stops at that terminal, returns an error, and does not execute any further terminals.



Common Properties

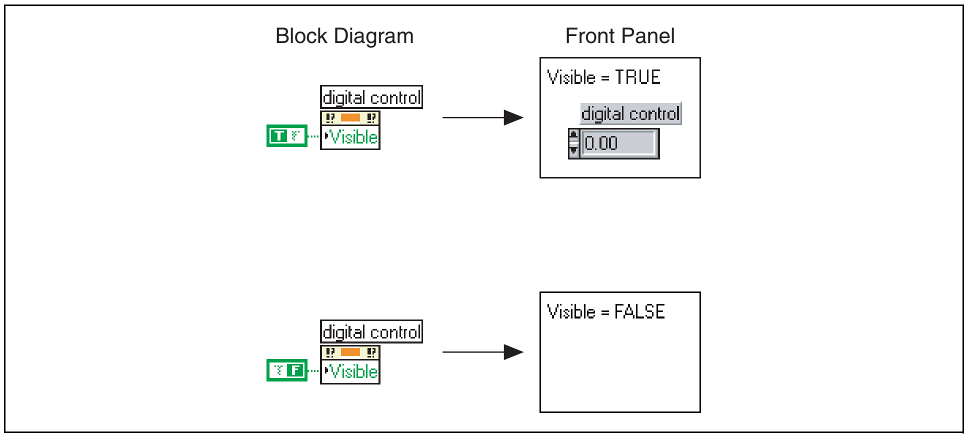
Many properties are available for front panel objects. This lesson describes the Visible, Disable, Key Focus, Blink, Position, Bounds, and Value properties that are common to all front panel objects. It also introduces some Property Nodes for specific kinds of controls and indicators.

Move the cursor over terminals in the Property Node to display more information about the property in the **Context Help** window. You also can right-click a property terminal and select **Help For *Property*** from the shortcut menu, where ***Property*** is the name of the property.

Visible Property

The Visible property writes or reads the visibility of a front panel object. The associated object is visible when TRUE, hidden when FALSE.

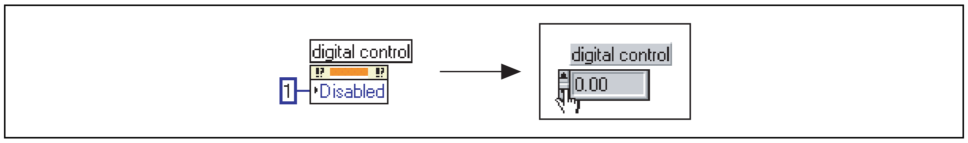
The following example sets the numeric control to an invisible state. A Boolean TRUE constant makes the control visible.



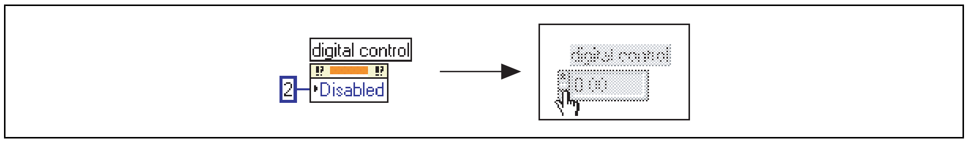
Disabled Property

The Disabled property writes or reads the user access status of an object. A value of `0` enables an object so that the user can operate it. A value of `1` disables the object, preventing operation. A value of `2` disables and dims the object.

The following example disables user access to the numeric control. The control does *not* change appearance when disabled.



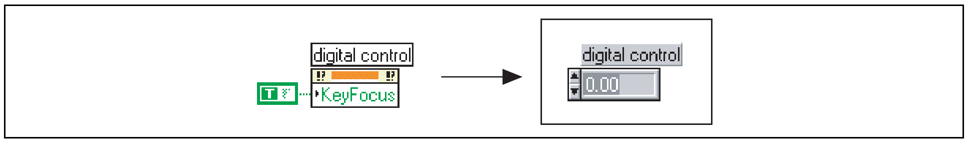
The following example disables user access to the numeric control and dims the control.



Key Focus Property

The Key Focus property writes or reads the key focus of a front panel object. When TRUE, the cursor is active in the associated object. On most controls, you can enter values into the control by typing them on the keyboard. You also can set the key focus on the front panel by pressing the <Tab> key while in run mode or by pressing the hot key associated with the control (assigned using the **Key Navigation** option).

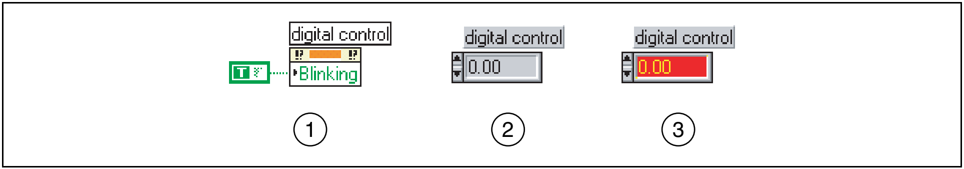
The following example makes the numeric control the key focus. You then can enter a new value in the control without selecting it with the cursor.



Blinking Property

The Blinking property reads or writes the blink status of an object. When this property is set to TRUE, an object begins to blink. You can set the blink rate and colors by selecting **Tools»Options** and selecting the **Front Panel** and **Colors Options** pages from the top pull-down menu. When this property is set to FALSE, the object stops blinking.

The following example enables blinking for the numeric control.

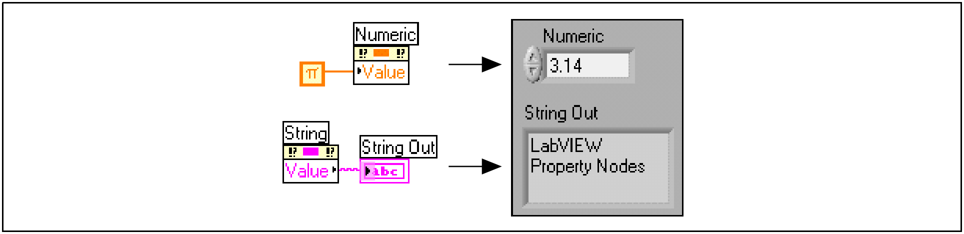


1 Block Diagram 2 Front Panel (normal) 3 Front Panel (blinking)

Value Property

The Value property reads or writes the current value of an object. When you set the Value property to write, it writes the wired value to an object whether it is a control or indicator. When you set the Value property to read, it reads the current value in either a control or indicator. Use the Value property sparingly. Use dataflow wiring techniques rather than Property Nodes to update the values of front panel objects.

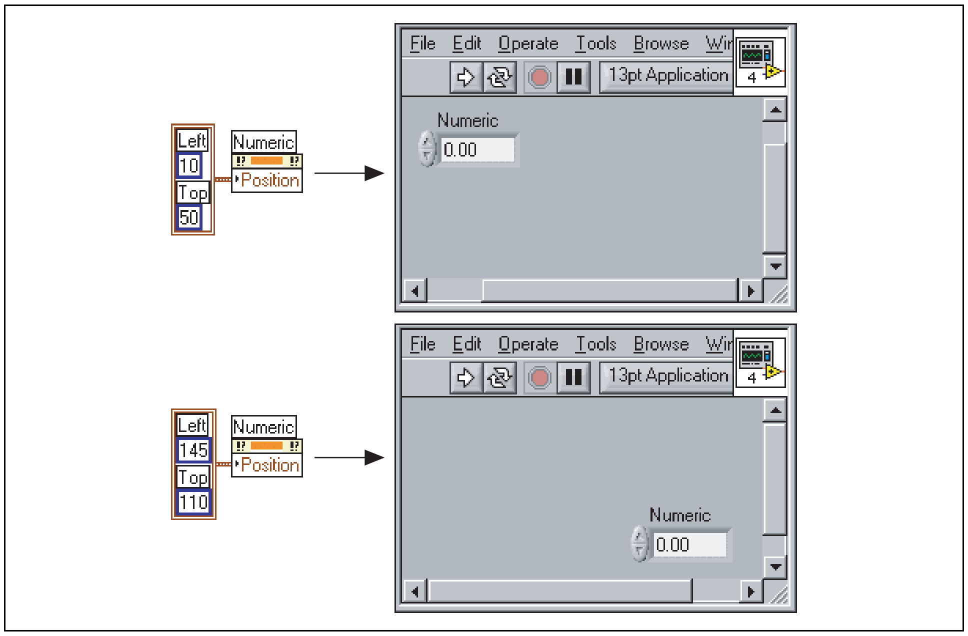
The following example shows a value of *pi* written to a numeric control and the value of one string being written to another string.



Position Property

The Position property sets or writes the position of an object's upper left corner on the front panel. The position is determined in units of pixels relative to the upper left corner of the front panel. This property consists of a cluster of two unsigned long integers. The first item in the cluster, **Left**, is the location of the left edge of the control relative to the left edge of the front panel, and the second item in the cluster, **Top**, is the location of the top edge of the control relative to the top edge of the front panel.

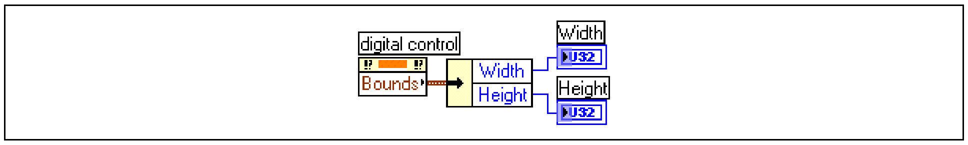
The following example changes the location of the numeric control on the front panel.



Bounds Property

The Bounds property reads the boundary of an object on the front panel in units of pixels. The value includes the control and all of its parts, including the label, legend, scale, and so on. This property consists of a cluster of two unsigned long integers. The first item in the cluster, **Width**, is the width of the object in pixels, and the second item in the cluster, **Height**, is the height of the object in pixels. This is a read-only property. It does not resize a control or indicator on the front panel. Some objects have other properties for resizing, such as the Plot Area Size property for graphs and charts.

The following example determines the bounds of the numeric control.



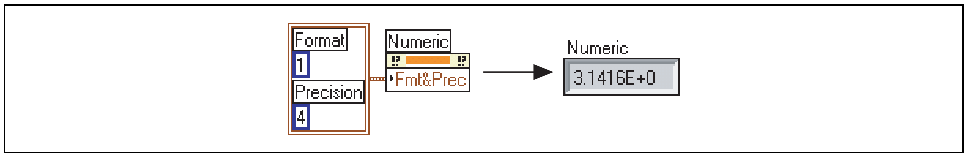
Numeric Property: Format and Precision

The Format and Precision property sets or writes the format (type of notation) and precision (number of digits displayed after the decimal point) of numeric front panel objects. The input is a cluster of two unsigned byte integers. The first element sets the format and the second sets the precision. The Format property can be one of the following integer values:

0	Decimal Notation
1	Scientific Notation

2	Engineering Notation
3	Binary Notation
4	Octal Notation
5	Hexadecimal Notation
6	Relative Time Notation

The following example sets the format of the numeric control to scientific notation and the precision to four digits.

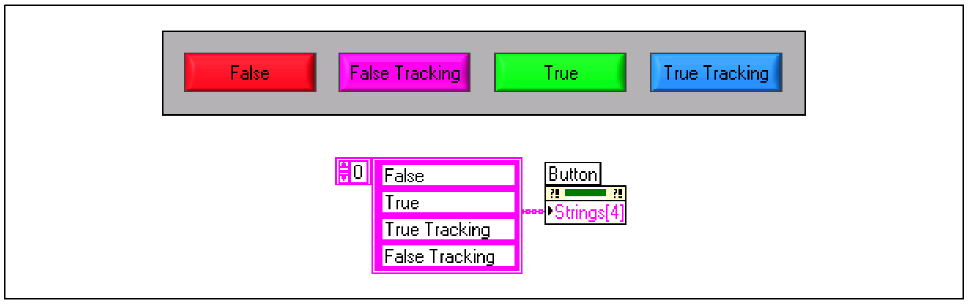


Boolean Property: Strings [4]

The Strings [4] property writes or reads the labels on a Boolean control. The input is an array of four strings that correspond to the False, True, True Tracking, and False Tracking states of a Boolean object.

- **True and False**—On and Off states of the Boolean object.
- **True and False Tracking**—Temporary transition levels between the Boolean states. True Tracking is the transition state when the Boolean object is changed from True to False. Tracking applies only to Boolean objects with **Switch When Released** and **Latch When Released** mechanical actions. These mechanical actions have a transitional state until you release the mouse. The text strings `True Tracking` and `False Tracking` are displayed during the transitional state.

The following example sets the display strings for the Switch control to the string choices Stop, Run, Stop? and Run?.

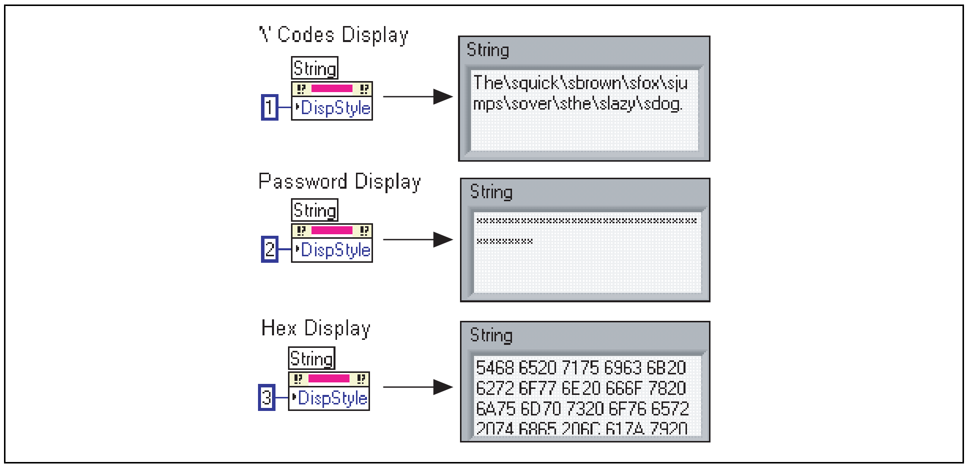


String Property: Display Style

The Display Style property writes or reads the display for a string control or indicator. An unsigned long integer determines the display mode.

0	Normal Display
1	‘\’ Codes Display
2	Password Display
3	Hex Display

The following example shows the text in the string control in the three other modes.



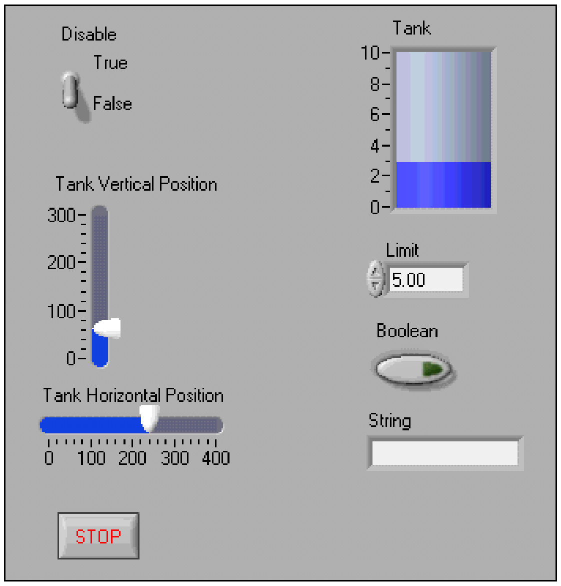
Exercise 3-1 Property Node Exercise VI

Objective: To build a VI that uses Property Nodes to manipulate common characteristics of panel objects.

In this exercise, you build a VI that programmatically changes the position, disabled, and color properties of front panel objects.

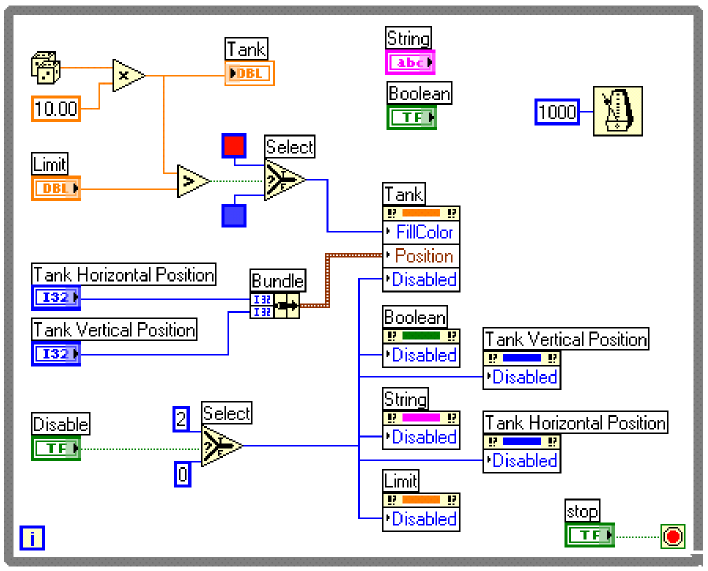
Front Panel

1. Open a new VI and build the following front panel.



Block Diagram

2. Open and build the following block diagram.



a. Place the While Loop, located on the **Functions»Execution Control** palette, on the block diagram. This structures the VI to continue running until the user presses the **Stop** button.

b. Place the Wait Until Next ms Multiple function, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. This function causes the While Loop to execute once a second. Create the constant by right-clicking the input terminal and selecting **Create»Constant**. Type `1000` into the constant.

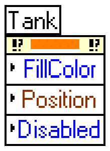
c. Place the Random Number (0-1) function, located on the **Functions»Arithmetic & Comparison»Express Numeric** palette, on the block diagram. This function creates a random number between zero and one.

d. Place the Multiply function, located on the **Functions»Arithmetic & Comparison»Express Numeric** palette, on the block diagram. This function multiplies two numbers together and is used here to scale the random number to be between zero and 10.

e. Place the Greater? function, located on the **Functions»Arithmetic & Comparison»Express Comparison** palette, on the block diagram. This function compares two values—in this case the random value and the limit value, and returns a value of TRUE if the random value is greater than the limit. Otherwise it returns a value of FALSE.

f. Place the Select function, located on the **Functions»Arithmetic & Comparison»Express Comparison** palette, on the block diagram. You will use two of these functions. This function takes a Boolean input and outputs the top value if the Boolean object is TRUE and the bottom value if the Boolean object is FALSE.

g. Place the Color Box Constant, located on the **Functions»All Functions»Numeric»Additional Numeric Constants** palette, on the block diagram. This constant colors the panel objects through their Property Node. You will need two of these constants. Click the constant with the Operating tool to display the color picker. Set one constant to red and set the other to blue.

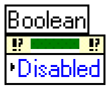


h. Create the Tank Property Node on the block diagram. Right-click the **Tank** terminal and select **Create»Property Node** from the shortcut menu. Use the Positioning tool to resize the node to show three terminals. Select the properties shown at left by right-clicking each terminal and selecting the item from the **Properties** menu. Right-click the Property Node and select **Change All To Write** from the menu.

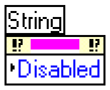
Tip You also can click a Property Node terminal with the Operating tool to display the **Properties** menu.

i. Right-click the **Tank Horizontal Position** and **Tank Vertical Position** controls and select **Representation»I32** from the shortcut menu.

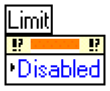
j. Place the Bundle function, located on the **Functions»All Functions»Cluster** palette, on the block diagram. This function clusters together the **Tank Horizontal Position** and **Tank Vertical Position** controls into the Position property for the tank.



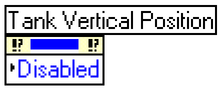
k. Create the Boolean Property Node on the block diagram. Right-click the **Boolean** terminal and select **Create»Property Node** from the shortcut menu. Select the Disabled property shown by right-clicking the terminal and selecting it from the **Properties** menu. Right-click the Property Node and select **Change To Write** from the shortcut menu.



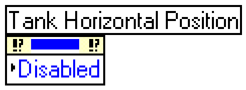
l. Create the String Property Node on the block diagram. Right-click the **String** terminal and select **Create»Property Node** from the shortcut menu. Select the Disabled property shown by right-clicking the terminal and selecting it from the **Properties** menu. Right-click the Property Node and select **Change To Write** from the menu.



m. Create the Limit Property Node on the block diagram. Right-click the **Limit** terminal and select **Create»Property Node** from the shortcut menu. Select the Disabled property shown by right-clicking the terminal and selecting it from the **Properties** menu. Right-click the Property Node and select **Change To Write** from the menu.



n. Create the Tank Vertical Position Property Node on the block diagram. Right-click the **Tank Vertical Position** terminal and select **Create»Property Node** from the shortcut menu. Select the Disabled property shown by right-clicking the terminal and selecting it from the **Properties** menu. Right-click the Property Node and select **Change To Write** from the menu.



o. Create the Tank Horizontal Position Property Node on the block diagram. To create this node, right-click the **Tank Horizontal Position** terminal and select **Create»Property Node** from the shortcut menu. Select the Disabled property shown by right-clicking the terminal and selecting it from the **Properties** menu. Right-click the Property Node and select **Change To Write** from the menu.

3. Save this VI as `Property Node Exercise.vi` in the `C:\Exercises\LabVIEW Basics II` directory.
4. Display the front panel and run the VI. Several things should be happening:
   - As the VI generates new random numbers and writes them to the tank, the fill color displays red if the random value is greater than the **Limit** value and the fill color displays blue if the random value is less than the **Limit**.
   - The two sliders change the position of the tank on the panel. Move these values and see how the tank moves.
   - The **Disable** switch controls whether you can change the values. Flip the **Disable** switch to True and all the panel objects except the **Disable** switch and the **Stop** button are dimmed and you cannot change their values.
5. Stop and close this VI when you are finished.

End of Exercise 3-1

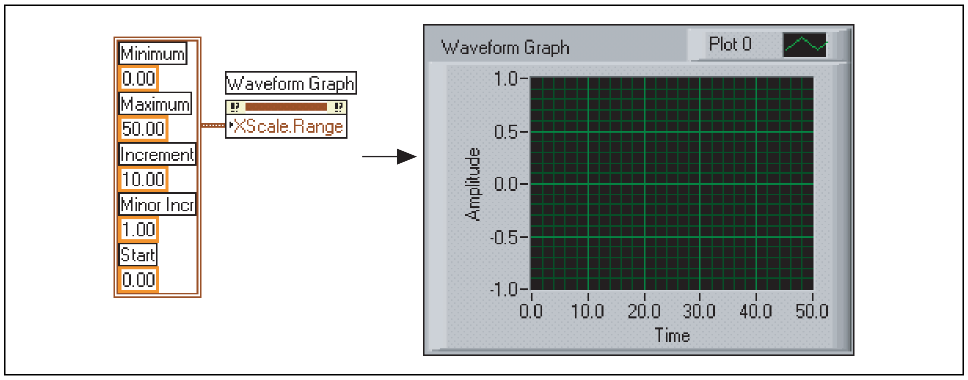
B. Graph and Chart Properties

You can use Property Nodes to control most graph and chart features, such as plot, background, grid colors, X and Y scale information, including minimum, maximum, and increment values, visibility of the legends and palette, plotting area size, cursors, and so on.

X (or Y) Range Property

The X (or Y) Range Property sets or reads the range and increment for the graph axis. The property accepts a cluster of five numeric values, depending on the data type of the graph or chart: minimum value, maximum value, major and minor increments between the axis markers, and the start value of the scale. To create this property, select **X Scale»Range»All Elements** from the property list. If there is not enough space to display all the increment values you have specified, LabVIEW selects an alternate increment value.

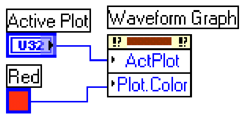
The following example sets the x-axis range to 0 to 50 with major axis increments of 10, minor increments of 1, and a start value of zero.



Active Plot and Plot Color Properties

The Active Plot and Plot Color properties set or read the active plot (the trace for which subsequent trace-specific properties are set or read) and the plot color for the active plot. Active Plot is an integer corresponding to the desired plot and Plot Color is an integer representing a color. The Plot Color property is accessed by selecting **Plot»Plot Color** from the property list.

The following example sets the color of the active plot using a Color Box constant set to red. When selecting the active plot, the Active Plot terminal must precede the Plot Color terminal on the Property Node.

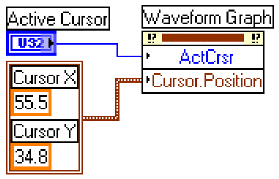


Active Cursor, Cursor Position, and Cursor Index Properties

The Active Cursor, Cursor Position, and Cursor Index properties set or read the active cursor, the position of that cursor on the graph, and the index (x-axis position) in the plot where the cursor resides. The Active Cursor property accepts an integer corresponding to the desired cursor when there is more than one cursor on the graph. Cursor Position consists of a cluster of two floating-point numbers representing the X and Y positions on the plot. To create the Cursor Position property, select **Cursor»Cursor Position»All Elements** from the property list. Cursor Index accepts an integer corresponding to an element in the array (plot). To access the Cursor Index property, select **Cursor»Cursor Index** from the property list.

For example, you can move a graphic selector on the plot, or you can lock a cursor to the plot or float it on the plotting surface. Values from each cursor are returned to an optional display on the front panel.

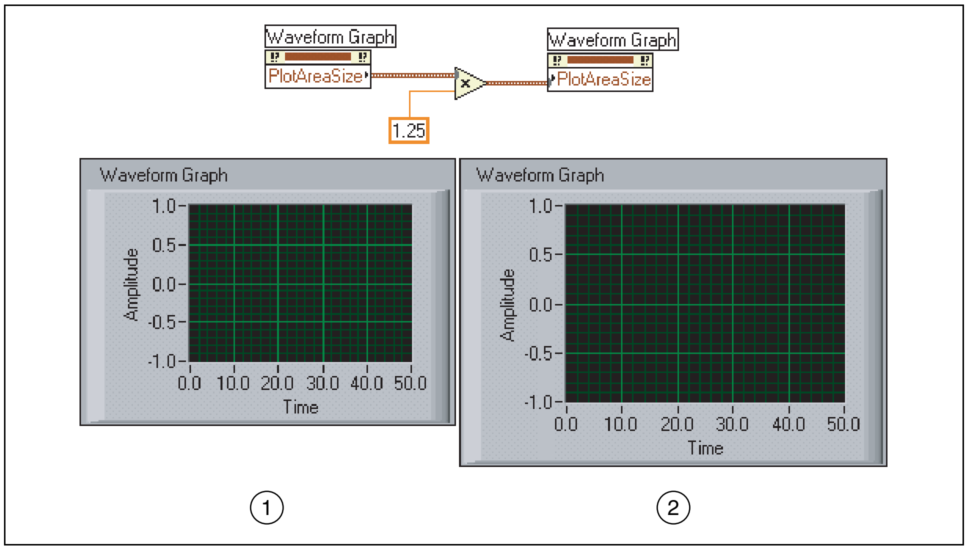
The following example places a cursor at position (55.5, 34.8). When selecting the cursor, the Active Cursor terminal must precede the Cursor Position terminal. Because the node executes from top to bottom, you can set another cursor's location by adding another set of Active Cursor and Cursor Position properties to this node.



Plot Area»Size Property

To read or change the size of a graph or chart plotting area, send new Width and Height values to the Plot Area Size property. The Width and Height are in units of screen pixels. For example, if you want to have a plot appear in a small window during part of your application, but appear larger later, you can use this property in conjunction with other properties that change the size of the front panel.

The following example resizes a graph to increase its width and height by 25%.



1 Before Execution	2 After Execution

Exercise 3-2 Temperature Limit VI

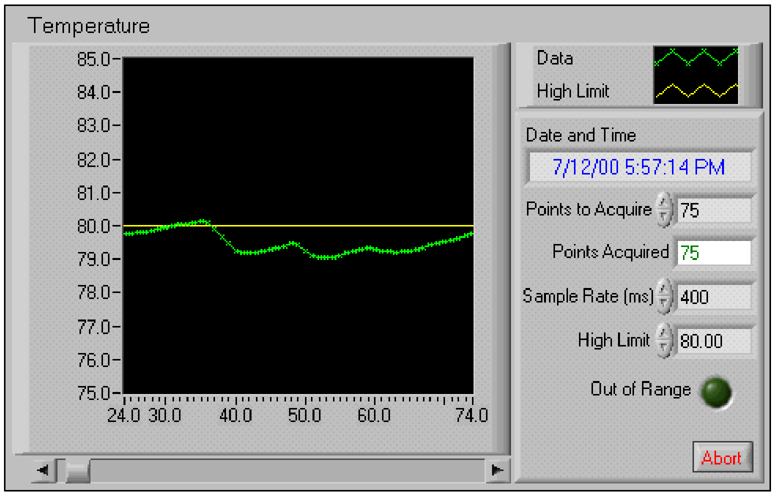
Objective: To create a VI that uses Property Nodes to clear a waveform chart and notify the user if the data exceed a limit.

You will finish building a VI that uses Property Nodes to perform the following tasks:

- Set the delta X value of the chart to the sample rate in seconds.
- Clear the waveform chart so it initially contains no data.
- Change the color of a plot if the data exceed a certain value.
- Make an alarm indicator blink if the data exceed a certain value.

Front Panel

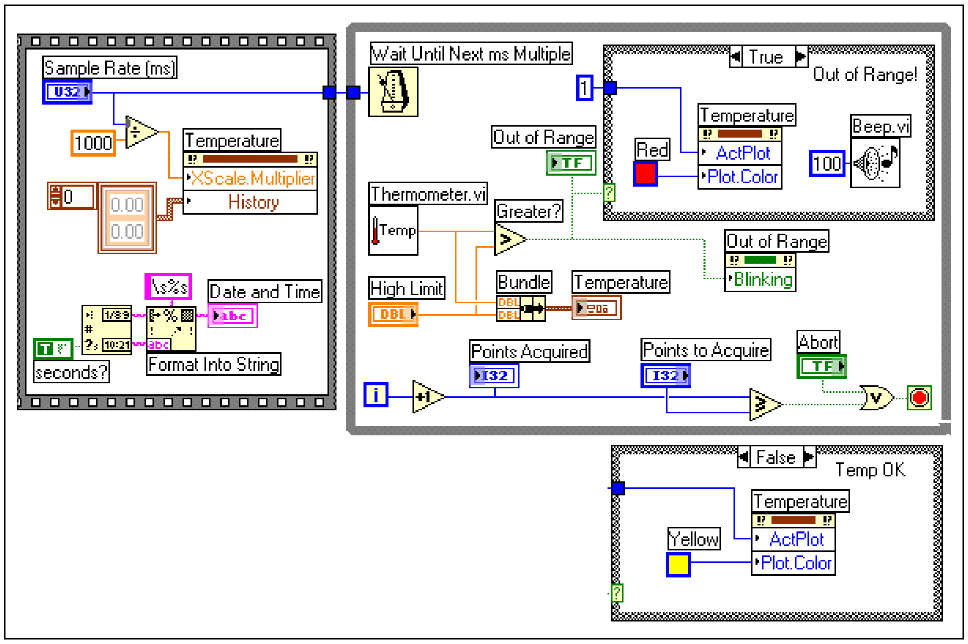
1. Open the Temperature Limit VI located in the `C:\Exercises\LabVIEW Basics II` directory. The front panel and a portion of the block diagram are already built for you.



Date and Time 7/12/00 5:57:14 PM
Points to Acquire 75
Points Acquired 75
Sample Rate (ms) 400
High Limit 80.00
Out of Range

Block Diagram

2. Open the block diagram.



3. Modify the VI so that it sets the delta X value to the sample rate and clears the Temperature chart before starting. While the VI acquires data, it should turn the **High Limit** trace red when the temperature exceeds the limit value, and the **Out of Range** LED should blink.

   a. In the Sequence structure, right-click the Temperature chart and select **Create»Property Node** from the shortcut menu to create a Property Node. Resize the node to two terminals. Select the Multiplier property, **X Scale»Offset and Multiplier»Multiplier**, and the History Data property.

      To clear a waveform chart from the block diagram, send an empty array of data to the History Data property. Right-click the History Data property and select **Create»Constant**. Make sure that the array constant is empty.

   b. In the Case structure inside the While Loop, create a Property Node for the Temperature chart that has two terminals. Create the Active Plot property and the Plot Color property, **Plot»Plot Color**. You will need these properties in both cases of the Case structure. Because the High Limit plot is plot 1, set the Active Plot property to one before setting the Plot Color property. If the data are greater than the High Limit, set the plot color to red. Otherwise, the plot color should be yellow. Use a Color Box constant to send the color value to the Plot Color property.

   c. Right-click the **Out of Range** LED to create the Blink Property Node. The LED should blink when the temperature is greater than the High Limit.

4. Save the VI.
5. Display the front panel and run the VI to confirm that it behaves correctly. Save and close the VI.

End of Exercise 3-2

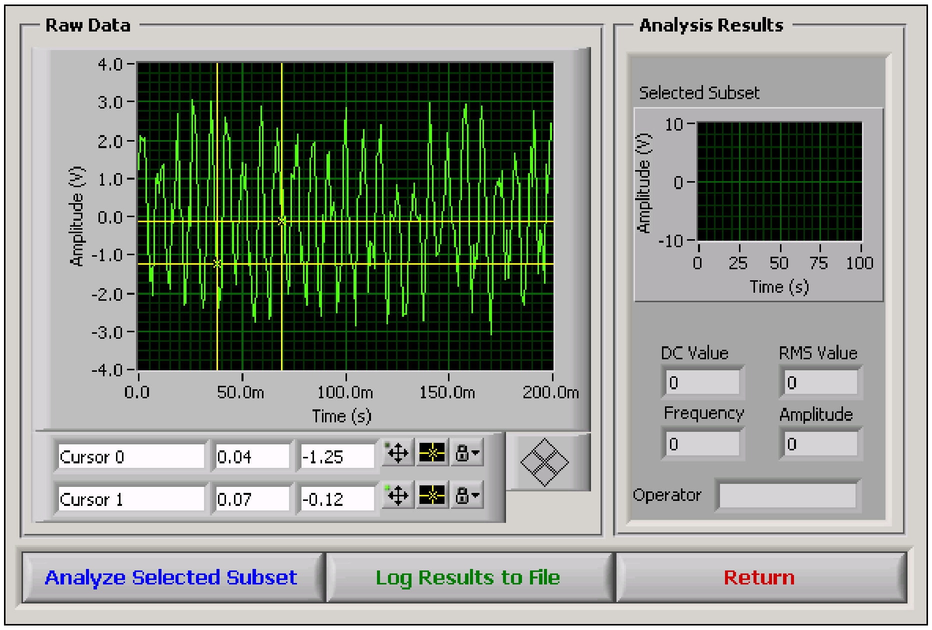
Exercise 3-3 Analyze & Present Data VI

Objective: To use Property Nodes with graph cursors.

Create a VI in which you use graph cursors to select a subset of data for analysis. In a later exercise, you will build a subVI that saves the results to disk.

Front Panel

1. Open the Analyze & Present Data VI located in the `C:\Exercises\LabVIEW Basics II` directory. The front panel is already complete.



Use the two cursors in the plot window to select a subset of data to analyze. A cursor can move freely or be locked to the plot. You control this action using the **Lock Control** button at the far right side of the cursor display.

Movement locked to the plot points.

Movement unrestricted in the plot window.

As you use the cursors you can use the Cursor Movement Selector to move the cursors with the Cursor Mover.

Cursor Movement Selector turned off.

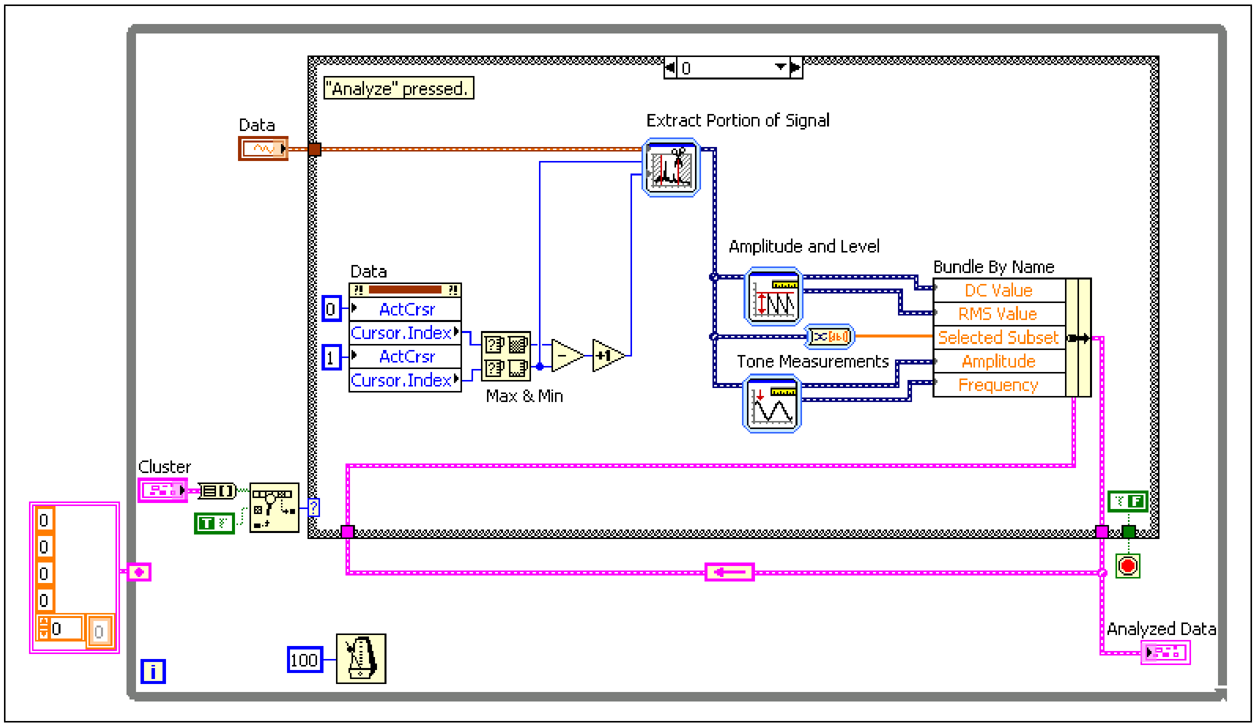
Cursor Movement Selector turned on. The Cursor Mover can move the cursor.

Use the Cursor Mover to move the cursors.

Use cursors that are locked to the plot. When the user clicks the **Analyze Selected Subset** button, the VI reads the location of each cursor and uses this information to find the DC, RMS, frequency, and amplitude values of the subset of data.

Block Diagram

2. Open and complete the following block diagram as described in steps 3 through 13.

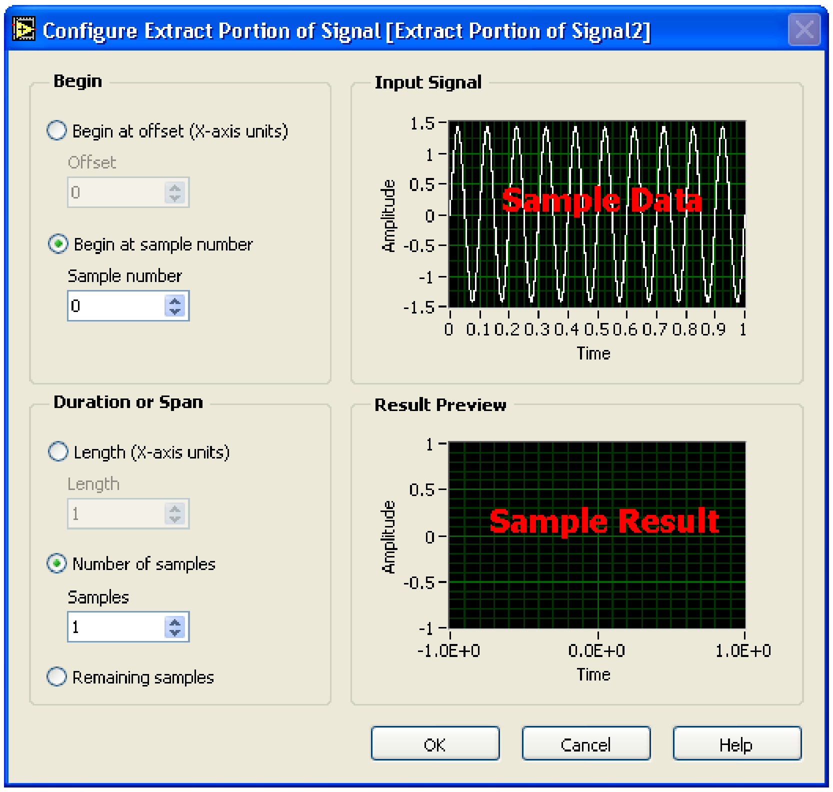


3. Right-click the Data terminal and select **Create»Property Node** from the shortcut menu to create the Data Property Node. Resize the Property Node to four terminals. Right-click the node and select the Active Cursor and **Cursor»Cursor Index** properties from the **Properties** menu. Right-click each Active Cursor property and select **Change to Write** from the shortcut menu.

   Property Nodes execute from top to bottom. The Property Node selects each cursor individually and returns the index of each cursor.

4. Place the Max & Min function, located on the **Functions»All Functions»Comparison** palette, on the block diagram. This function helps determine the beginning and ending index of the cursor locations.

5. Place the Extract Portion of Signal Express VI, located on the **Functions»Signal Manipulation** palette, on the block diagram. Complete the following steps to configure the **Extract Portion of Signal** dialog box that appears.



Configure Extract Portion of Signal [Extract Portion of Signal2]

Begin

( ) Begin at offset (X-axis units)

Offset: 0

(●) Begin at sample number

Sample number: 0

Duration or Span

( ) Length (X-axis units)

Length: 1

(●) Number of samples

Samples: 1

( ) Remaining samples

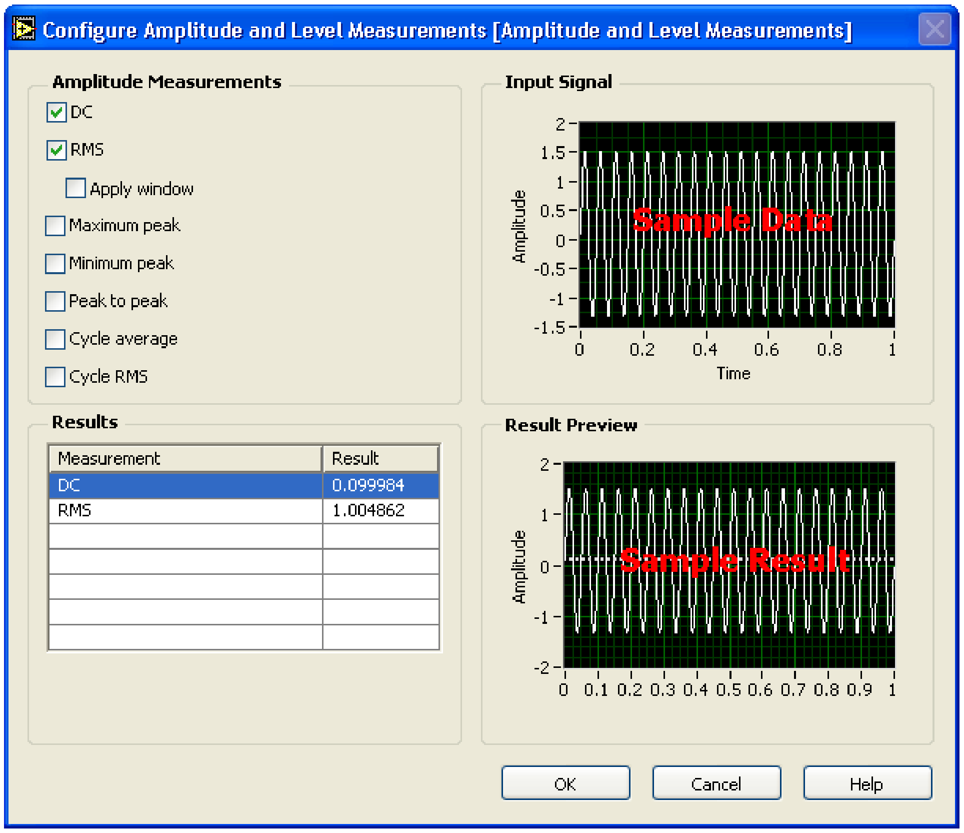
a. Set **Begin** to **Begin at sample number**.

b. Set **Duration or Span** to **Number of samples** so you can pass the range of samples that you want to extract.

c. Click the **OK** button to return to the block diagram.

d. Right-click the Extract Portion of Signal Express VI and select **View As Icon** from the shortcut menu to conserve space on the block diagram.

6. Place the Amplitude and Level Measurements Express VI, located on the **Functions»Signal Analysis** palette, on the block diagram. Complete the following steps to configure the **Amplitude and Level Measurements** dialog box that appears.



Configure Amplitude and Level Measurements [Amplitude and Level Measurements]

Amplitude Measurements

- ✓ DC
- ✓ RMS
  - ☐ Apply window
- ☐ Maximum peak
- ☐ Minimum peak
- ☐ Peak to peak
- ☐ Cycle average
- ☐ Cycle RMS

Results

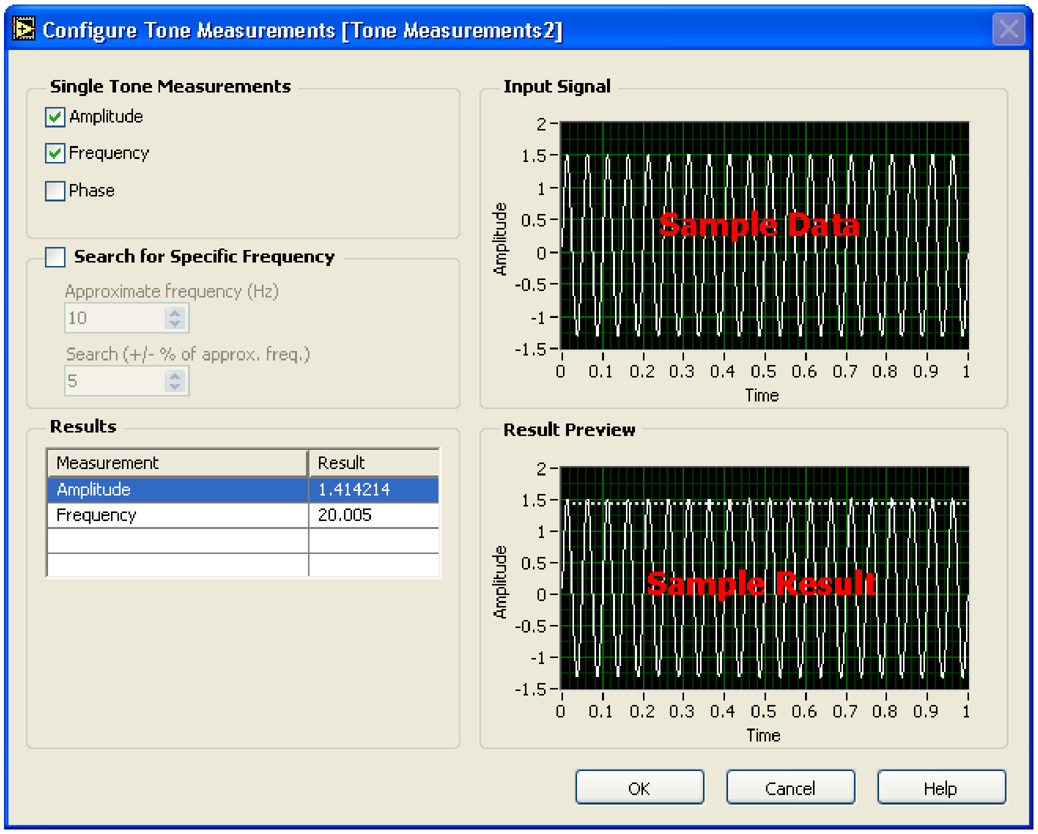
Measurement	Result
DC	0.099984
RMS	1.004862

a. Place checkmarks in the **DC** and **RMS** amplitude measurements checkboxes to output both the DC and RMS values of the input signal.

b. Click the **OK** button to return to the block diagram.

c. Right-click the Amplitude and Level Measurements Express VI and select **View As Icon** from the shortcut menu to conserve space on the block diagram.

7. Place the Tone Measurements Express VI, located on the **Functions» Signal Analysis** palette, on the block diagram. Complete the following steps to configure the **Tone Measurements** dialog box that appears.


Configure Tone Measurements [Tone Measurements2]

Single Tone Measurements
- [✓] Amplitude
- [✓] Frequency
- [ ] Phase

[ ] Search for Specific Frequency

Approximate frequency (Hz): 10

Search (+/- % of approx. freq.): 5

Results

Measurement	Result
Amplitude	1.414214
Frequency	20.005

a. Place checkmarks in the **Amplitude** and **Frequency** checkboxes to output both the Amplitude and Frequency values of the input signal.

b. Click the **OK** button to return to the block diagram.

c. Right-click the Tone Measurements Express VI and select **View As Icon** from the shortcut menu to conserve space on the block diagram.

8. Place the Feedback Node, located on the **Functions»All Functions» Structures** palette, on the block diagram. Feedback Nodes transfer values from one loop iteration to the next and are synonymous with Shift Registers. Use the Feedback Node to avoid unnecessarily long wires in loops. The Initializer terminal, shown at left, appears on the left side of the While Loop when you place the Feedback Node and allows you to place a known value into the Feedback Node during code initialization.

Note Feedback Nodes can only be placed inside For Loops or While Loops.

9. Wire the empty cluster constant on the left of the While Loop to the Feedback Node Initializer terminal.

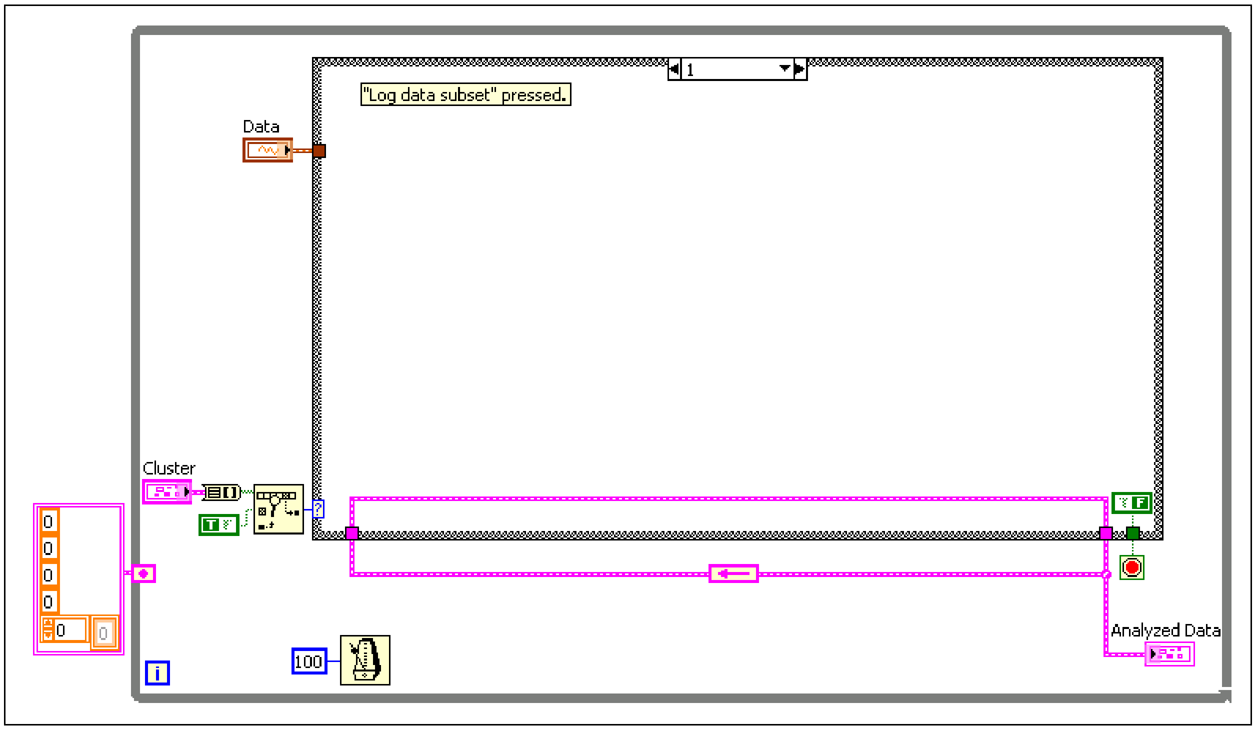
10. Place the Bundle by Name function, located on the **Functions»All Functions»Cluster** palette, on the block diagram. Wire the output of the Feedback Node to the input cluster terminal, which is the middle terminal of the Bundle by Name function.

    Resize the Bundle by Name function to have five inputs. If the label names are not correct, right-click the name and select the correct item from the **Select Item** shortcut menu.

11. Wire the outputs of the Amplitude and Levels Express VI and Tone Measurements Express VI to the Bundle by Name function.

12. Wire the output of the Extract Portion of Signal Express VI to the Bundle by Name function. Because the output of the Extract Portion of Signal Express VI is a dynamic data type, the Convert from Dynamic Data Express VI, shown at left, automatically appears when you connect the wire to the Bundle by Name function.

13. Finish the VI so that data passes from the Feedback Node unchanged through the other cases. Make sure the other cases in the Case structure pass the data through correctly, straight through the cases as shown in the following block diagram.



14. Save the VI. You will use this VI later in the course.
15. Display the front panel and run the VI. Move the cursors along the graph to select a subset of data to analyze and click the **Analyze Selected Subset** button. The results appear in the **Analysis Results** cluster. When you have finished, click the **Return** button.
16. Close the VI when you are finished.

End of Exercise 3-3

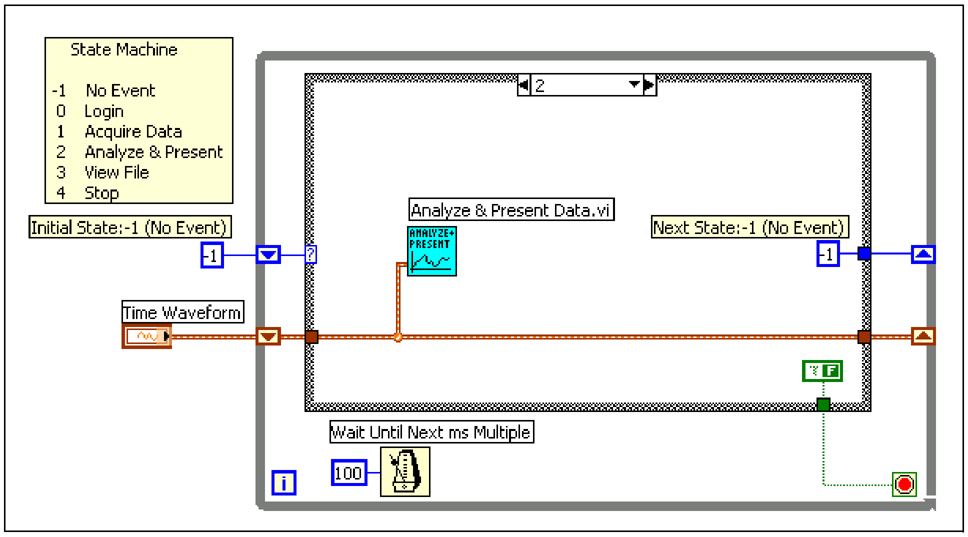
Exercise 3-4 State Machine with Analyze & Present Data VI

Objective: To add the Analyze & Present Data VI to the application started in Exercise 1-1.

Add the Analyze & Present Data VI you created in Exercise 3-3 to the project.

Block Diagram

1. Open the State Machine with Enhanced Acquire Data VI you created in Exercise 2-5. The front panel is already complete. Open the block diagram.



2. Display case 2 of the Case structure and delete the One Button Dialog function. Select **Functions»All Functions»Select a VI** and navigate to the `C:\Exercises\LabVIEW Basics II` directory to add the Analyze & Present Data VI you completed in Exercise 3-3 to this case.

Note If you did not complete Exercise 3-3, use the Analyze & Present Data VI located in the `C:\Solutions\LabVIEW Basics II` directory.

3. Wire the waveform containing the collected data to the **Data** input.
4. Select **File»Save As** to save the VI as `State Machine with Analyze & Present Data.vi` in the `C:\Exercises\LabVIEW Basics II` directory.
5. Display the front panel and run the VI. Make sure you can acquire the data and analyze a selected subset of the data.
6. Stop and close the VI when you are finished.

End of Exercise 3-4

C. Control References

If you are building a VI that contains several Property Nodes or if you are accessing the same property for several different controls and indicators, you can place the Property Node in a subVI and use control references to access that node. A control reference is a refnum to a specific front panel object. This lesson shows one way to use control references. Refer to the *LabVIEW User Manual* for more information about control references.

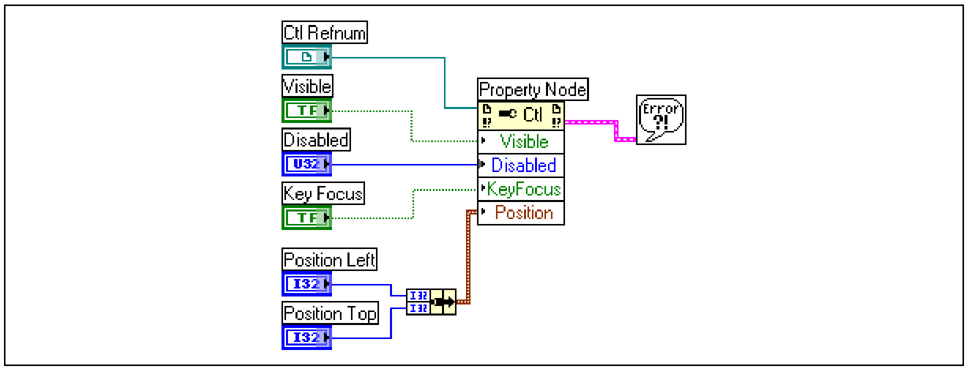
Creating Control References

To create a control reference for a front panel object, right-click the object or its block diagram terminal and select **Create»Reference** from the shortcut menu.

You can wire this control reference to a subVI that contains Property Nodes. However, the subVI connector pane must contain a control refnum terminal.

Using Control References

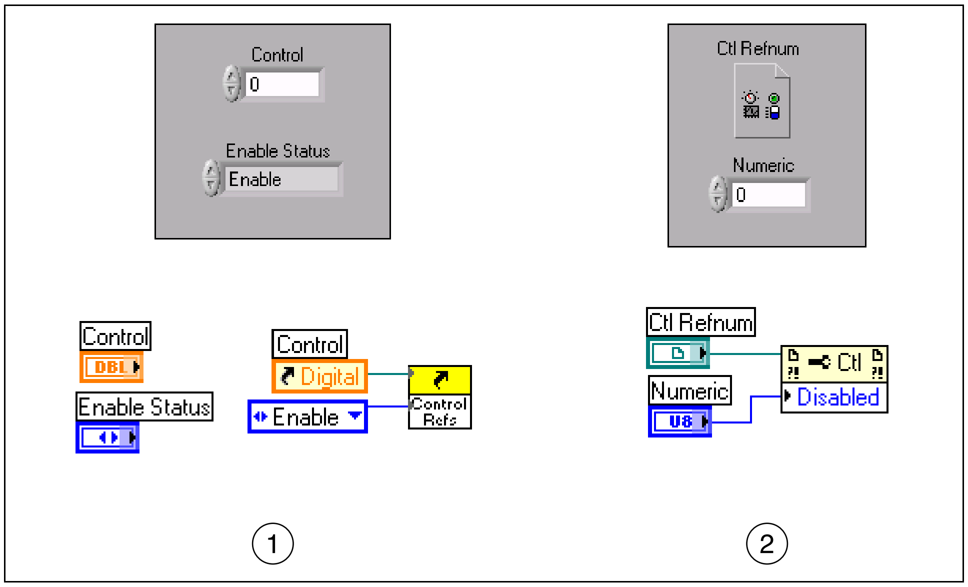
The following block diagram shows how you can use a control reference to set many of the properties for a front panel object.



Notice the appearance of the Property Node in the block diagram. This type of Property Node is located on the **Functions»All Functions»Application Control** palette. It is not linked to a control until a Control Refnum is wired to its Refnum input. The advantage of this type of Property Node is its generic nature. Because it has no explicit link to any one control, it may be reused for many different controls.

Setting properties with the control reference method is useful for working with properties that apply to all controls, such as the Disabled property. Some properties are only applicable to certain controls, such as the Housing Size property which only applies to Dial and Knob controls.

The following example shows how to construct a VI that uses a control reference on the SubVI to determine the Enable/Disable state of a control on the Main VI front panel.



1 Main VI 2 SubVI

The Main VI sends a reference for the digital numeric control to the SubVI along with a value of zero, one, or two from the enumerated control. The SubVI receives the reference by means of the Ctl Refnum on its front panel. The reference is then passed to the Property Node. Because the Property Node now has a link back to the digital numeric control in the Main VI, it can change properties of that control. In this case the Enabled/Disabled state is manipulated.

Strictly Typed and Weakly Typed Control References

Strictly typed control refnums accept only control refnums of the same kind of data. For example, if the type of a strictly typed control refnum is a slide of 32-bit integers, you can wire a slide of 32-bit integers, a slide of 8-bit integers, or a slide of double-precision scalars to the control refnum terminal, but not a slide of a cluster of 32-bit integers.

Control refnums that you create from a control are strictly typed by default. A red star in the lower left corner of the control refnum on the front panel indicates the control refnum is strictly typed. On the block diagram, `(strict)` appears on the Property Node or Invoke Node wired to the control refnum terminal to indicate that the control refnum is strictly typed.

Note Because the latch mechanical actions are incompatible with strictly typed control refnums, Boolean controls with latch mechanical action produce weakly typed control refnums.

Weakly typed control refnums are more flexible in the type of data they accept. For example, if the type of a weakly typed control refnum is slide, you can wire a 32-bit integer slide, single-precision slide, or a cluster of 32-bit integer slides to the control refnum terminal. If the type of a weakly typed control refnum is control, you can wire a control refnum of any type of control to the control refnum terminal.

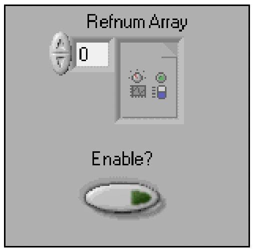
Exercise 3-5 Disable Controls VI and Control Refs Example VI

Objective: To build a VI that uses control references to access Property Nodes from a subVI.

Build a subVI that accesses an array of control references and assigns the Disabled property. Then modify the Property Node Exercise VI you built in Exercise 3-1 to use the subVI rather than the original Property Nodes.

Front Panel

1. Open a blank VI and build the following front panel.



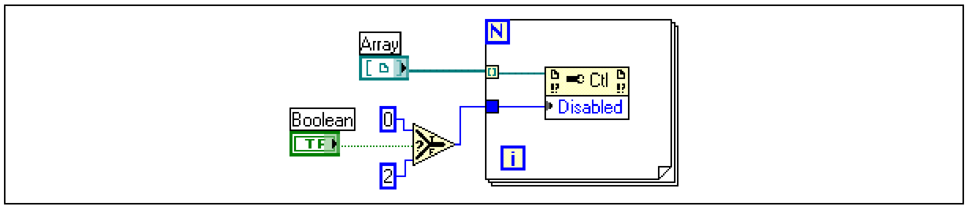
   a. Place an empty Array, located on the **Controls»All Controls» Array & Cluster** palette, on the front panel.

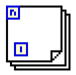
   b. Place a Control Refnum, located on the **Controls»All Controls» Refnum** palette, in the array.

   c. Place a Boolean Push Button on the front panel.

Block Diagram

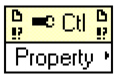
2. Open and build the following block diagram.





   a. Place a For Loop, located on the **Functions»All Functions» Structures** palette, on the block diagram. The For Loop is used to auto-index through the array of control refnums so that each refnum and property is handled separately.

   b. Place a Property Node, located on the **Functions»All Functions» Application Control** palette, on the block diagram. You use this Property Node as a generic control type. When you wire the Refnum Array to the refnum input terminal, the function changes slightly.

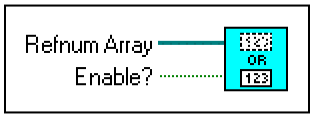


   c. Right-click the Property terminal and select **Properties»Disabled**. Right-click the terminal again and select **Change To Write**.

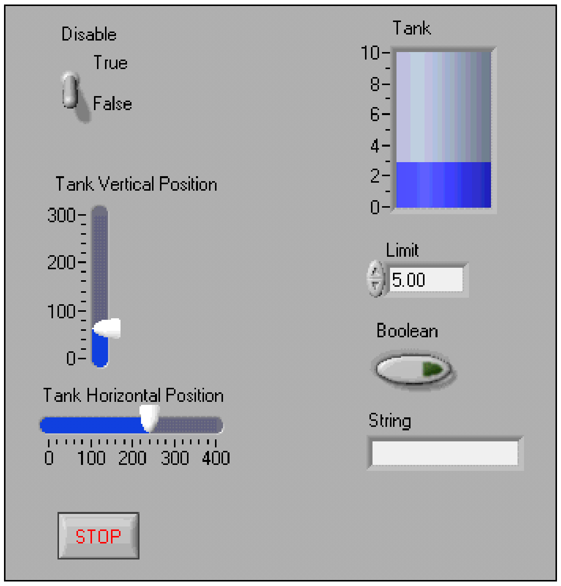
d. Place the Select function, located on the **Functions»Arithmetic & Comparison»Express Comparison** palette, on the block diagram. This function takes a Boolean input and outputs the top value of 0 (enabled) if the Boolean is True and the bottom value of 2 (disabled and dimmed) if the Boolean is False.

e. Place a Numeric constant, located on the **Functions»Arithmetic & Comparison»Express Numeric** palette, on the block diagram. You will need two of these with a value of 0 and 2 for the Select function.

3. Select **File»Save As** to save the VI as `Disable Controls.vi` in the `C:\Exercises\LabVIEW Basics II` directory. You will use this VI later in the course.
4. Return to the front panel. Build an icon and connector pane for the VI, similar to the following example.

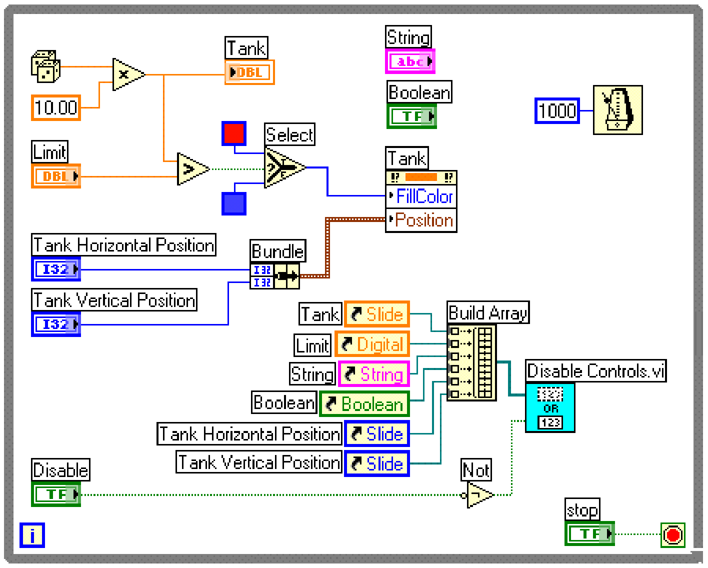


5. Save and close the VI.
6. Make a calling VI for the subVI you finished in step 5. Open the Property Node Exercise VI you built in Exercise 3-1. The front panel is already complete.



7. Select **File»Save As** and rename the VI `Control Refs Exercise.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

8. Open the block diagram and modify it as shown in the following example.



   a. Create Control References for the six controls by right-clicking their terminals and selecting **Create»Reference** from the shortcut menu. Delete the property nodes for all controls except Tank.

   b. Place the Disable Controls subVI, located on the **Functions»All Functions»Select A VI** palette, on the block diagram. This is the VI you finished in step 5.

   c. Place the Build Array function, located on the **Functions»All Functions»Array** palette, on the block diagram. Wire all the control references into this function and pass the output array to the Disable Controls subVI.

   d. Place the Not function, located on the **Functions»Arithmetic & Comparison»Express Boolean** palette, on the block diagram to invert the value going into the subVI.

9. Save the VI.

10. Display the front panel and run the VI. Notice that when you set the Disable switch to True, all the controls are dimmed as they were in the Property Node Exercise VI from Exercise 3-1.

11. Close the VI when you are finished.

End of Exercise 3-5

D. LabVIEW Type Definitions

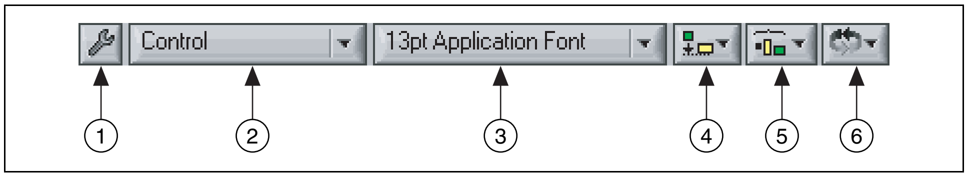
Custom Controls

You can customize LabVIEW controls and indicators to change their appearance on the front panel. You also can save these controls for use in other VIs. Programmatically, they function the same as standard LabVIEW controls. Refer to the *LabVIEW Custom Controls, Indicators, and Type Definitions* Application Note for more information about creating and using custom controls and type definitions.

Launch the Control Editor by selecting a control on the front panel and selecting **Edit»Customize Control**. The Control Editor appears with the selected front panel object in its window. The Control Editor has two modes, edit mode and customize mode.

Edit Mode

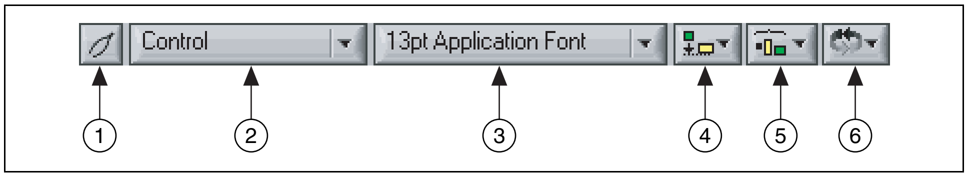
In edit mode, you can right-click the control and manipulate its settings as you would in the LabVIEW programming environment.



1 Edit Mode	3 Text	5 Distribute Objects
2 Type Definition Status	4 Align Objects	6 Reorder Objects

Customize Mode

In customize mode, you can move the individual components of the control around with respect to each other. For a listing of what you can manipulate in customize mode, select **Window»Show Parts Window**.



1 Customize Mode	3 Text	5 Distribute Objects
2 Type Definition Status	4 Align Objects	6 Reorder Objects

One way to customize a control is to change its type definition status. You can save a control as a control, a type definition, or a strict type definition, depending on the selection visible in the **Type Def. Status** ring. The control option is the same as a control you would select from the **Controls** palette. You can modify it in any way you need to, and each copy you make and change retains its individual properties.

Saving Controls

After creating a custom control, you can save it for use later. By default, controls saved on disk have a `.ctl` extension.

You also can use the Control Editor to save controls with your own default settings. For example, you can use the Control Editor to modify the defaults of a waveform graph, save it, and later recall it in other VIs.

Type Definition

A type definition control is a master copy of a custom control. All copies of this kind of custom control must be of the same data type. For example, if you create a type definition custom control having a numeric representation of Long, you cannot make a copy of it and change its representation to Unsigned Long. Use a type definition when you want to place a control of the same data type in many places. If you change the data type of the type definition in the Control Editor, the data type updates automatically in all VIs using the custom control. However, you can still individually customize the appearance of each copy of a type definition control.

A strict type definition control must be identical in all facets where it is used. In addition to data type, its size, color, and appearance also must be the same. Use a strict type definition when you want to have completely identical objects in many places and to modify all of them automatically. You can still have unique labels for each instance of a strict type definition.

When a type definition control is used, LabVIEW ensures that the data type is the same anywhere the control is used. A strict type definition is more restrictive and LabVIEW ensures that almost everything about the control remains the same.

A type definition can have a different name, description, default value, size, color, or style of control, such as a knob instead of a slide, as long as the data type matches the master copy of the control. A type definition only identifies the correct type for each instance of a custom control. The type does not include things like data range for numeric controls, or item names in a ring control. If you change the data range on a numeric control or an item name on a ring control that are part of a type definition control, these properties do not change on all instances of the control. However, if you change the item name in a type definition for an enumerated type control, all instances change as well, since the item name is part of the type for an enumerated type control. You also can use property nodes with type definition controls as you would with a standard control.

A strict type definition forces almost everything about the control to be identical, including its size, color, and appearance. Strict type definitions are more restrictive, and unlike General type definitions, they define other values, such as range checking on numeric controls and item names on ring controls. The only flexibility to a strict type definition is the name, description, and default value which all can be different for separate instances of the a strict type definition control. The only properties available for a strict type definition control are those that affect the appearance of the control such as Visible, Disabled, Key Focus, Blinking, Position, and Bounds. For example, if you have a strict type definition which is made up of a cluster of various controls, properties for each individual control would not be available. Only appearance properties for the overall custom control are customizable.

Type definitions and strict type definitions are typically used to create a custom control using a cluster of many controls. If you need to add a new control and pass a new value to every subVI, you can add the new control to the custom control cluster, instead of having to add the new control to each subVIs front panel and making new wires and terminals.

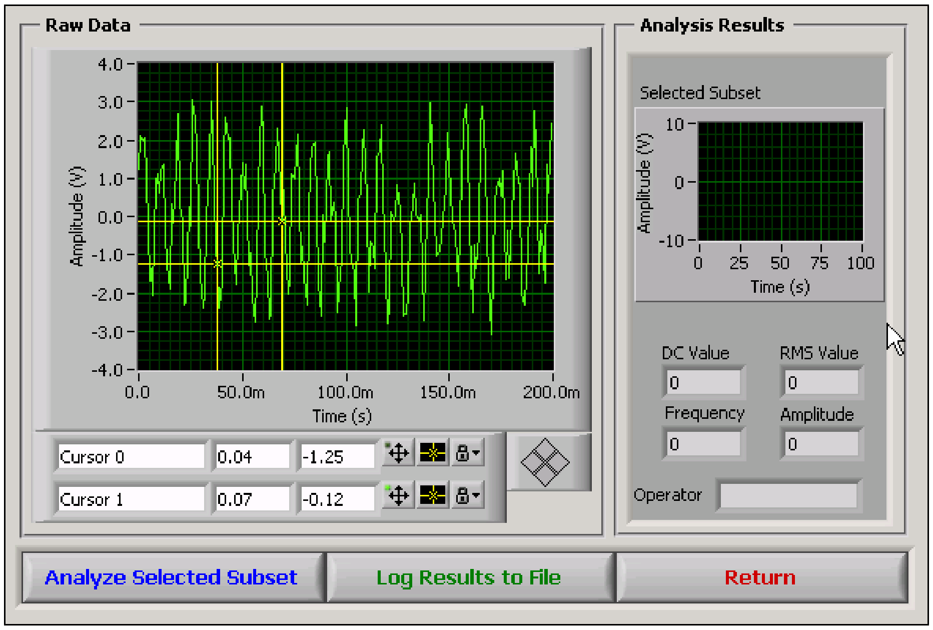
Exercise 3-6 Analyze & Present Data with TypeDef VI

Objective: To modify the Analyze & Present Data VI to use a TypeDef control.

In the application you are developing, you want to build around a program architecture that includes the ability to easily edit the code in the future. Because the Analyze & Present Data VI uses a cluster to store the analyzed data, the complexity of editing the code increases. In this exercise you convert the cluster constant and cluster into a strict type definition to allow you to easily add new data items to the cluster in the future without having to rewrite your code.

Front Panel

1. Open the Analyze & Present Data VI you completed in Exercise 3-3, located in the `C:\Exercises\LabVIEW Basics II` directory.



2. Right-click the border of the cluster in Analysis Results as shown in the previous front panel and select **Advanced»Customize** from the shortcut menu to open the cluster in the Control Editor.

   a. In the **Type Def. Status** pull-down menu, select **Type Def.** for the type definition.

   b. Select **File»Save** and save the type definition control as `Extracted Data.ctl` in the `C:\Exercises\LabVIEW Basics II` directory.

   c. Close the Control Editor. When prompted to replace the current control with the newly created control, select **Yes**.

Block Diagram

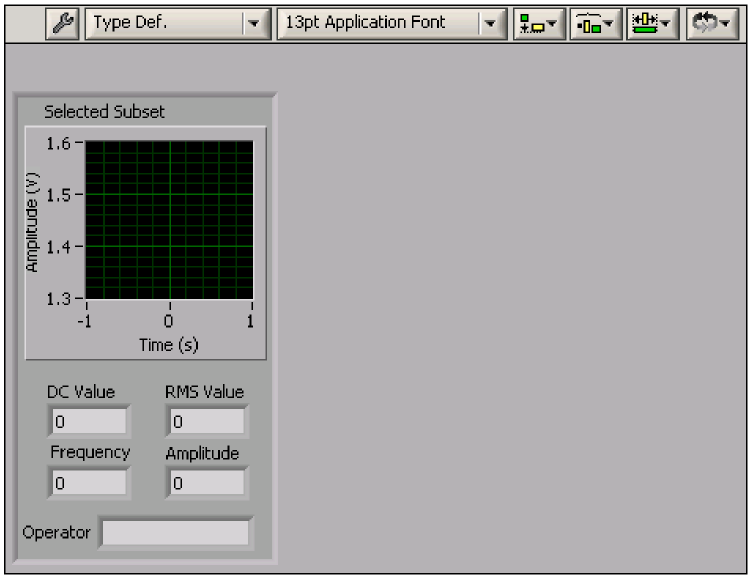
3. Display the block diagram and replace the cluster constant wired to the Feedback Node Initializer terminal with the custom control. Right-click the border of the cluster constant, select **Replace»All Controls»Select a VI** from the shortcut menu, select `Extracted Data.ctl` in the `C:\Exercises\LabVIEW Basics II` directory and click the **Open** button.

   When you modify the type definition custom cluster, all other clusters connected to it automatically update.

4. Add a string indicator to the type definition custom cluster.

   a. Right-click the cluster constant and select **Open Type Def.** from the shortcut menu to open `Extracted Data.ctl` in the Control Editor.

   b. Add a string indicator to the cluster as shown in the following example. Label the string indicator `Operator`.



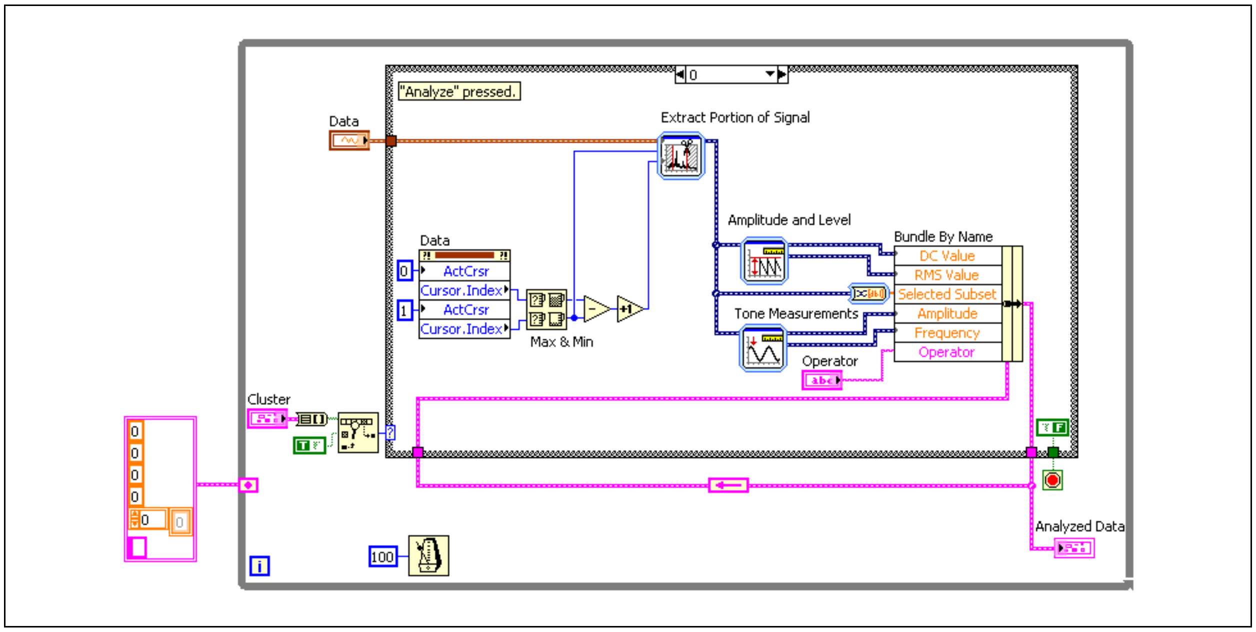
   c. Select **File»Save** to save the changes to the control.

   d. Select **File»Apply Changes** to update all controls and constants that are connected to the type definition.

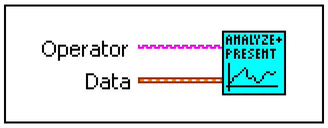
   e. Close the Control Editor.

5. Notice that a string indicator has been added to the cluster constant wired to the Feedback Node Initializer terminal on the block diagram. Display the front panel and notice that a string indicator has been added to the cluster indicator.

6. Display the block diagram and resize the Bundle by Name function to add an input for the Operator string. Right-click the input and select **Create»Control** from the shortcut menu. The following block diagram should result.



7. Display the front panel. Add the **Operator** control to a terminal on the connector pane as shown in the following example.



In a later exercise, you pass information about the Operator into this VI.

8. Save and close the VI.

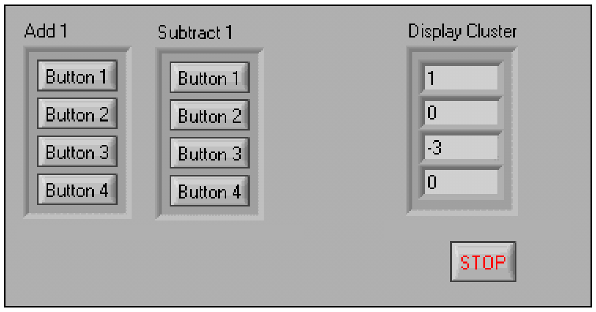
End of Exercise 3-6

Summary, Tips, and Tricks

- Property Nodes are powerful tools for expanding user interface capabilities. You use Property Nodes to programmatically manipulate the appearance and functional characteristics of front panel controls and indicators.
- You create Property Nodes from the **Create** shortcut menu of a control or indicator or a terminal for a control or indicator on the block diagram.
- You can create several Property Node terminals for a single front panel object, enabling you to configure properties from several locations in the block diagram.
- You can set or read properties such as user access (enable, disable, dim), colors, visibility, location, and blinking.
- Property Nodes greatly expand graph and chart functionality by changing plot size, adjusting the scale values, and operating or reading cursors.
- Use the **Context Help** window to learn more about individual properties.
- You can use control references and refnums to access Property Nodes inside subVIs.
- Use type definitions and strict type definitions to link all the instances of a custom control or indicator to a master definition so you can make changes to all instances by editing only the master definition.

Additional Exercises

3-7 Modify Exercise 1-2 located in the `C:\Exercises\LabVIEW Basics II` directory, by adding a cluster of four labeled buttons with mechanical action set to **Latch when released**. Each time you click a button in the new cluster, decrement the display by one. Use the following front panel to get started. Select **File»Save As** to save the VI as `Cluster Example 2.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

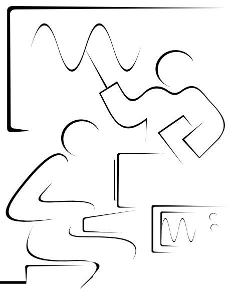


3-8 Build a VI that manipulates a button control on the front panel. The button should control the execution of the VI (that is, terminate the While Loop). Use a Dialog Ring to select the following options: **Enable and Show** button, **Disable** button, **Dim** button, and **Hide** button. Use Property Nodes to implement the actions that the Dialog Ring specifies. Test the different modes of operation for the button while trying to stop the VI. Select **File»Save As** to save the VI as `Button Input.vi` in the `C:\Exercises\LabVIEW Basics II` directory when you are finished.

3-9 Open the Analyze & Present Data VI you completed in Exercise 3-3 and modify the VI so that if the number of data points in the subset is one point or less, the **Log Results to File** button is dimmed. Select **File»Save As** and save the VI as `Analyze & Present Data with Dimming.vi` in the `C:\Exercises\LabVIEW Basics II` directory when you are finished.

Notes

Lesson 4
Local and Global Variables



This lesson describes techniques for passing data without the use of wires. This concept is powerful and must be used carefully in VIs. The first section of this lesson discusses data management. Local variables are then introduced, followed by global variables. The end of the lesson describes when variables should and should not be used in VIs.

You Will Learn:

A. About data management techniques in LabVIEW

B. How to use local variables

C. How to use global variables

D. Some tips about using local and global variables

A. Data Management Techniques in LabVIEW

LabVIEW follows a dataflow model for running VIs. A block diagram node executes when all its inputs are available. When a node completes execution, it supplies data to its output terminals and passes the output data to the next node in the dataflow path.

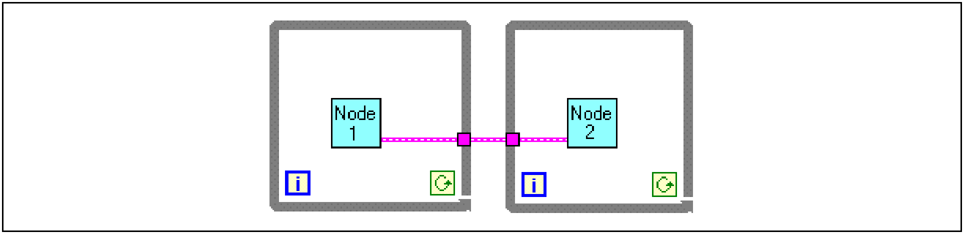
Visual Basic, C++, JAVA, and most other text-based programming languages follow a control flow model of program execution. In control flow, the sequential order of program elements determines the execution order of a program.

In LabVIEW, because the flow of data rather than the sequential order of commands determines the execution order of block diagram elements, you can create block diagrams that have simultaneous operations.

Block diagram nodes not connected by wires can execute in any order. You can use a sequence structure to control execution order when natural data dependency does not exist. You also can create an artificial data dependency in which the receiving node does not actually use the data received. Instead, the receiving node uses the arrival of data to trigger its execution.

Exchanging Data between Parallel Block Diagrams

Do not use wires to connect nodes if you need to exchange data between block diagrams that run in parallel. Parallel block diagrams can be two parallel loops on the same block diagram or two VIs that are called without any data flow dependency. The following block diagram does not run the two loops in parallel because of the wire between the two subVIs.



The wire creates a data dependency because the second loop does not start until the first loop finishes and passes the data through its tunnel. To make the two loops run concurrently, remove the wire. To pass data between the subVIs, use another technique, such as a local or global variable.

B. Local Variables

A front panel object has only one block diagram terminal, and your application might need to access the data in that terminal from more than one location. Use local variables to access front panel objects from more than one location in a single VI and to pass data between block diagram structures that you cannot connect with a wire.

Creating Local Variables

Right-click an existing front panel object or block diagram terminal and select **Create»Local Variable** from the shortcut menu to create a local variable. A local variable icon for the object appears on the block diagram.

You also can select **Functions»All Functions»Structures»Local Variable** and place a local variable on the block diagram. You then must associate the local variable node, shown at left, with a control or indicator. Right-click the local variable node and select a front panel object from the **Select Item** shortcut menu. The shortcut menu lists all the front panel objects that have owned labels. You also can use the Operating tool or Labeling tool to click the local variable node and select the front panel object from the shortcut menu. LabVIEW uses owned labels to associate local variables with front panel objects, so label the front panel controls and indicators with descriptive owned labels.

Read and Write Variables

After you create a local or global variable, you can read data from the variable or write data to it. Refer to Section C, *Global Variables*, of this lesson for more information about global variables.

By default, a new variable receives data. This kind of variable works as an indicator and is a write local or global. When you write new data to the local or global variable, the associated front panel control or indicator updates to the new data.

You also can configure a variable to behave as a data source, or a read local or global. Right-click the variable and select **Change To Read** from the shortcut menu to configure the variable to behave as a control. When this node executes, the VI reads the data in the associated front panel control or indicator.

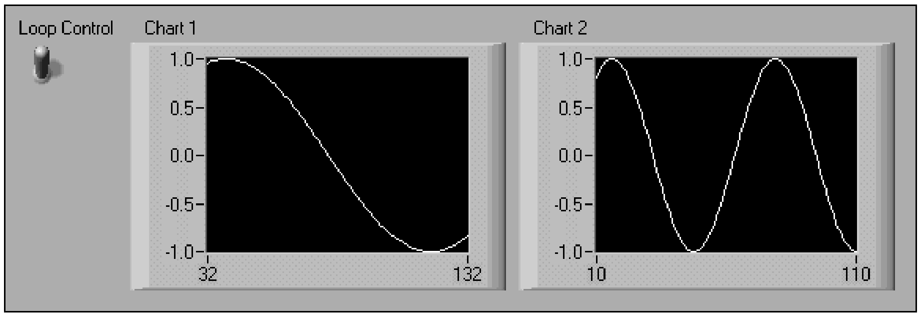
To change the variable to receive data from the block diagram rather than provide data, right-click the variable and select **Change To Write** from the shortcut menu.

On the block diagram, you can distinguish read locals or globals from write locals or globals the same way you distinguish controls from indicators.

A read local or global has a thick border similar to a control. A write local or global has a thin border similar to an indicator.

Local Variable Example

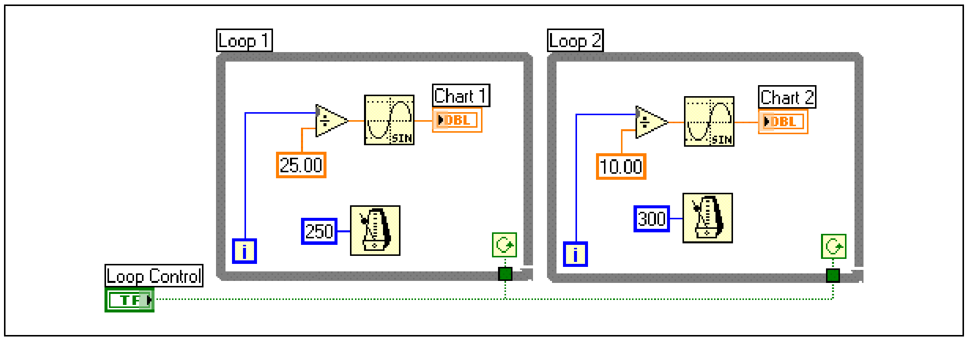
You could use a local variable associated with a single front panel switch to control two parallel While Loops, as shown in the following front panel.



Parallel While Loops are not connected by a wire, and they execute simultaneously. The first two methods shown below will not work as desired. The third method, which uses a local variable, works correctly.

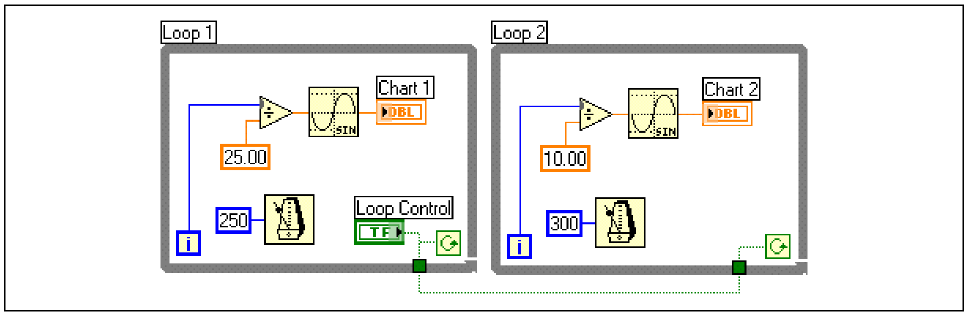
Method 1 (Incorrect)

Place the **Loop Control** terminal outside of both loops and wire it to each conditional terminal, as shown in the following block diagram. The **Loop Control** terminal is read only once, before either While Loop begins executing, because LabVIEW is a dataflow language and the status of the Boolean control is a data input to both loops. If TRUE is passed to the loops, the While Loops run indefinitely. Turning off the switch does not stop the VI because the switch is not read during the iteration of either loop.



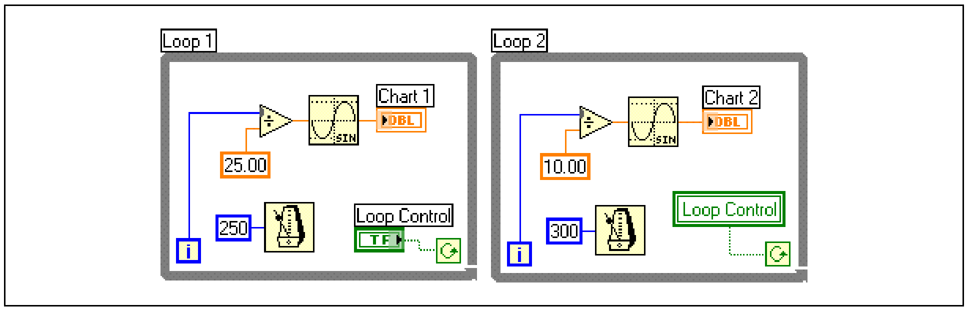
Method 2 (Incorrect)

Move the **Loop Control** terminal inside Loop 1 so that it is read in each iteration of Loop 1, as shown in the following block diagram. Although Loop 1 terminates properly, Loop 2 does not execute until it receives all its data inputs. Loop 1 does not pass data out of the loop until the loop stops, so Loop 2 must wait for the final value of the **Loop Control**, available only after Loop 1 finishes. Therefore, the loops do not execute in parallel. Also, Loop 2 executes for only one iteration because its conditional terminal receives a FALSE value from the **Loop Control** switch in Loop 1.



Method 3 (Correct)

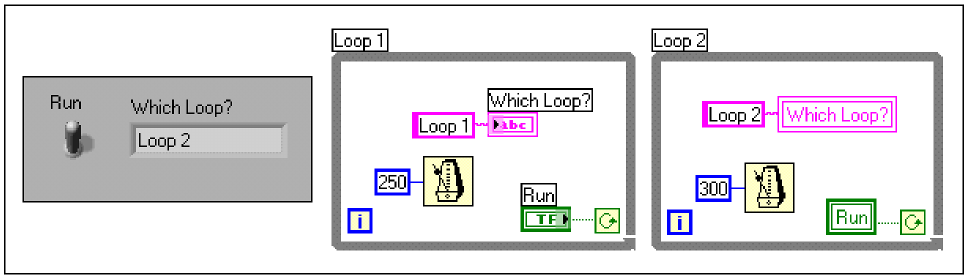
Loop 2 reads a local variable associated with the switch, as shown in the following block diagram. When you set the switch to FALSE on the front panel, the switch terminal in Loop 1 writes a FALSE value to the conditional terminal in Loop 1. Loop 2 reads the **Loop Control** local variable and writes a FALSE to the Loop 2 conditional terminal. Thus, the loops run in parallel and terminate simultaneously when you turn off the single front panel switch.



With a local variable, you can write to or read from a control or indicator on the front panel. Writing to a local variable is similar to passing data to any other terminal. However, with a local variable you can write to it even if it is a control or read from it even if it is an indicator. In effect, with a local variable, you can access a front panel object as both an input and an output.

The following example contains a single string indicator. If you want to update that indicator to display the loop that is currently executing, you must use a local variable because you can place the indicator terminal in only one loop.

However, using a local variable, you can access the same front panel indicator from more than one location on the block diagram, so that a single indicator displays the loop that is executing. The **Which Loop?** indicator is placed in Loop 1 and a local variable instance of that indicator is placed in Loop 2. This example shows how an indicator can be updated from two separate locations on the block diagram.



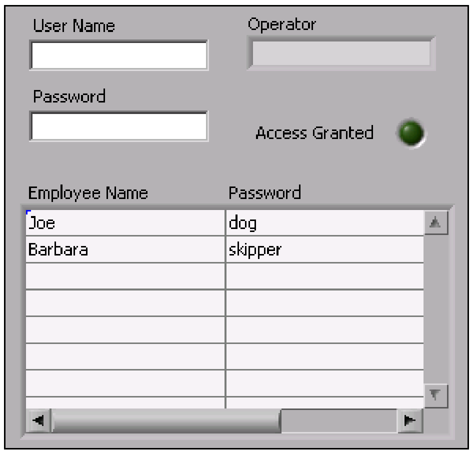
Exercise 4-1 Verify Information VI

Objective: To build a VI that demonstrates the Simple VI architecture.

Build a VI that accepts a name and password and checks for a match in a table of employee information. If the name and password match, the confirmed name and a verification Boolean object are returned.

Front Panel

1. Open a new VI and build the following front panel.

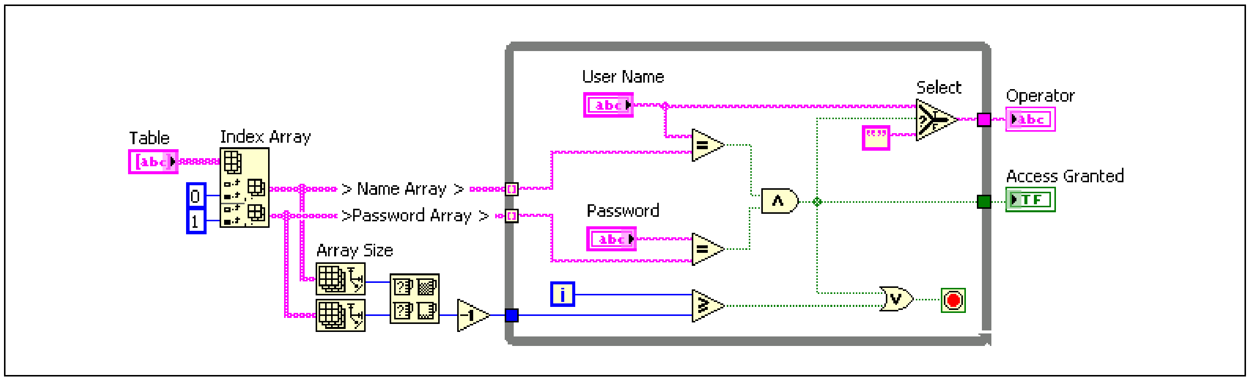


Employee Name	Password
Joe	dog
Barbara	skipper

2. Modify the User Name and Password string controls by right-clicking each control and selecting **Limit to single line** from the shortcut menu.
3. Modify the Password string control by right-clicking the control and selecting **Password Display** from the shortcut menu.
4. Place a Table control, located on the **Controls»All Controls»List & Table** palette, on the front panel. The table is a 2D array of strings where the first cell is at element 0,0.
    a. Resize the table to contain two columns.
    b. Right-click the Table and select **Visible Items»Label** to hide the label.
    c. Add free labels for the **Employee Name** and **Password** columns.
5. Enter the information shown in the previous figure in the table and save those values as default by right-clicking the table and selecting **Data Operations»Make Current Value Default** from the shortcut menu.

Block Diagram

6. Open and build the following block diagram.



a. Place a While Loop, located on the **Functions»Execution Control** palette, on the block diagram. This structures the VI to continue checking for a match until a name match occurs or there are no more rows in the table.

b. Place the Index Array function, located on the **Functions»All Functions»Array** palette, on the block diagram. This function extracts the column of names, and the column of passwords from the table. When you wire the table to the array input of this function, two indices—rows and columns—appear.

- Right-click the bottom index, **columns**, and select **Create» Constant** from the shortcut menu. This extracts the name information from column 0 of the table.
- Resize the function to add another input terminal.
- Right-click the bottom index, **columns**, and select **Create» Constant** from the shortcut menu. This extracts the password information from column 1 of the table.

c. Place the Array Size function, located on the **Functions»All Functions»Array** palette, on the block diagram. You use two of these functions to determine the size of the name array and password array.

d. Place the Max & Min function, located on the **Functions»All Functions»Comparison** palette, on the block diagram. This function determines which is the smallest array and uses that information to set the number of loop iterations.

e. Place the Decrement function, located on the **Functions» Arithmetic & Comparison»Express Numeric** palette, on the block diagram. This function decreases the number of names in the array by one to control the While Loop, which starts indexing at 0.

f. Place the Equal? function, located on the **Functions»Arithmetic & Comparison»Express Comparison** palette, on the block diagram. You use two of these functions to check if the **User Name** input and **Password** input match a table entry.

g. Place the Greater or Equal? function, located on the **Functions» Arithmetic & Comparison»Express Comparison** palette, on the block diagram. This function controls the While Loop conditional terminal. The loop continues to run while the current iteration number is less than the number of rows in the table.

h. Place the And function, located on the **Functions»Arithmetic & Comparison»Express Boolean** palette, on the block diagram. This function checks to see if the **User Name** and **Password** both match.

i. Place the Empty String constant, located on the **Functions»All Functions»String** palette, on the block diagram. If a match is not found an empty string is returned in the **Operator** indicator.

j. Place the Select function, located on the **Functions»Arithmetic & Comparison»Express Comparison** palette, on the block diagram. If the Password and Name match, the User Name is sent to the Operator indicator; otherwise, the Empty String is sent.

k. Place the Or function, located on the **Functions»Arithmetic & Comparison»Express Boolean** palette, on the block diagram. This function checks to see if a match has occurred or if the end of the array has been reached to stop the While Loop.

7. Enable indexing on each array entering the While Loop by right-clicking each tunnel and selecting **Enable Indexing** from the shortcut menu.

8. Select **File»Save As** to save the VI as `Verify Information.vi` in the `C:\Exercises\LabVIEW Basics II` directory. You use this VI later in the course.

9. Display the front panel and enter some names and passwords in the table control. The names do not have to be the same ones used in the course manual. If you have not already done so, right-click the table and select **Data Operations»Make Current Values Default** from the shortcut menu to permanently store the names and passwords in the table.

10. Type values in the User Name and Password controls and run the VI.

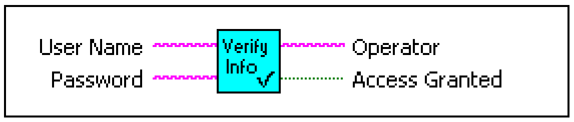
    If the User Name and Password match one of the rows in the table, the name returns in the Operator indicator and the **Access Granted** LED turns on. Otherwise, an empty string returns and the **Access Granted** LED remains off. Try several combinations of names and passwords to make sure the VI behaves correctly.

11. Stop the VI.

12. Create an icon for the VI so you can use it as a subVI in a later exercise. Right-click the icon in the top right corner of either the front panel or the block diagram and select **Edit Icon** from the shortcut menu. Design an icon similar to the following example.

Verify
Info

13. Create a connector pane for the VI by right-clicking the icon in the top right corner of the front panel and selecting **Show Connector** from the shortcut menu. Select the pattern and connect the front panel objects to the terminals as shown in the following connector pane.



   a. Right-click the connector pane and select **Patterns** from the shortcut menu. Select a pattern with at least four terminals, two each on the left and right sides of the connector.

   b. Use the Wiring tool to click a terminal in the connector pane, then click the corresponding front panel object to associate controls and indicators with the connector pane terminals.

14. Save and close the VI.

End of Exercise 4-1

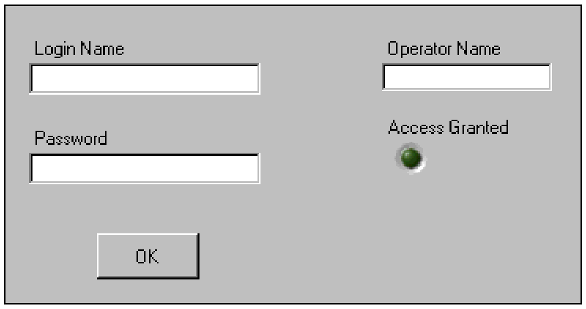
Exercise 4-2   Login VI

Objective: To use local variables to initialize controls on the front panel.

This VI demonstrates a good use of sequence structures and local variables for initializing code.

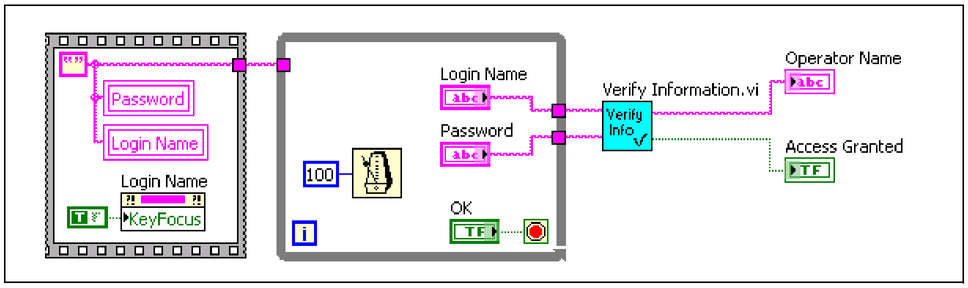
Front Panel

1. Open the Login VI located in the `C:\Exercises\LabVIEW Basics II` directory. The front panel of the VI is already built.



Block Diagram

2. Complete the following block diagram.



Notice that the local variables are enclosed in a single-frame sequence structure, and that the empty string constant is wired to the border of the While Loop. This ensures that both local variables are updated before the While Loop starts.

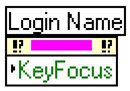
Login Name

a. Create the **Login Name** local variable by right-clicking the **Login Name** terminal and selecting **Create»Local Variable** from the shortcut menu. This variable is set to write local and resets the login name to an empty string.

Password

b. Create the **Password** local variable by right-clicking the **Password** terminal and selecting **Create»Local Variable** from the shortcut menu. This variable is set to write local and resets the password string to an empty string.

c. Place the Empty String constant, located on the **Functions»All Functions»String** palette, on the block diagram. This constant passes string values to the Login Name and Password local variables.

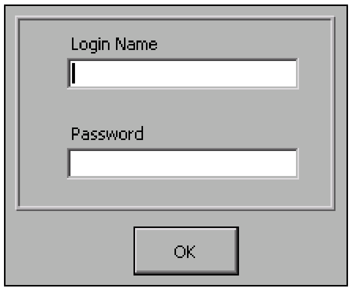


d. Create the Login Name Property Node by right-clicking the **Login Name** terminal and selecting **Create»Property Node** from the shortcut menu. Select the Key Focus property. Right-click the property and select **Change To Write** from the shortcut menu. Wire a TRUE Boolean constant to the Key Focus property.

e. Place the Verify Information VI you built in Exercise 4-1. Select **Functions»All Functions»Select a VI**, navigate to `C:\Exercises\LabVIEW Basics II`, and select the Verify Information VI to place it on the block diagram. This VI checks the name and password values for a match in a table of employee information.

Note If you have trouble wiring the string constant to a local variable, right-click the local and select **Change to Write Local** from the shortcut menu.

3. Save the VI.
4. Display the front panel and run the VI. Notice that the **Login Name** and **Password** controls reset to empty strings when you start the VI.
5. Resize the front panel to include only the necessary objects, and use a decoration to improve the appearance of the front panel, as shown in the following example.



6. Save and close the VI when you are finished.

End of Exercise 4-2

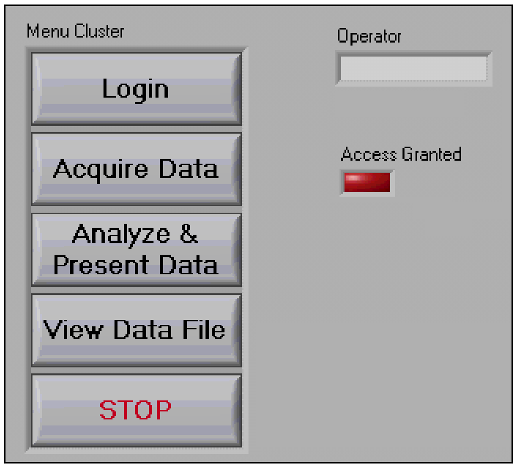
Exercise 4-3 State Machine with Login VI

Objective: To use the Login VI completed in Exercise 4-2 to provide password security to an application.

In this VI, you add more functionality to the final application. Add the Login VI you completed in Exercise 4-2 to the state machine and edit the VI so that until the user logs in with a correct user name and password, only the **Login** and **Stop** buttons are enabled.

Front Panel

1. Open and modify the front panel of the State Machine with Analyze & Present Data VI located in the `C:\Exercises\LabVIEW Basics II` directory. You created this VI in Exercise 3-4.



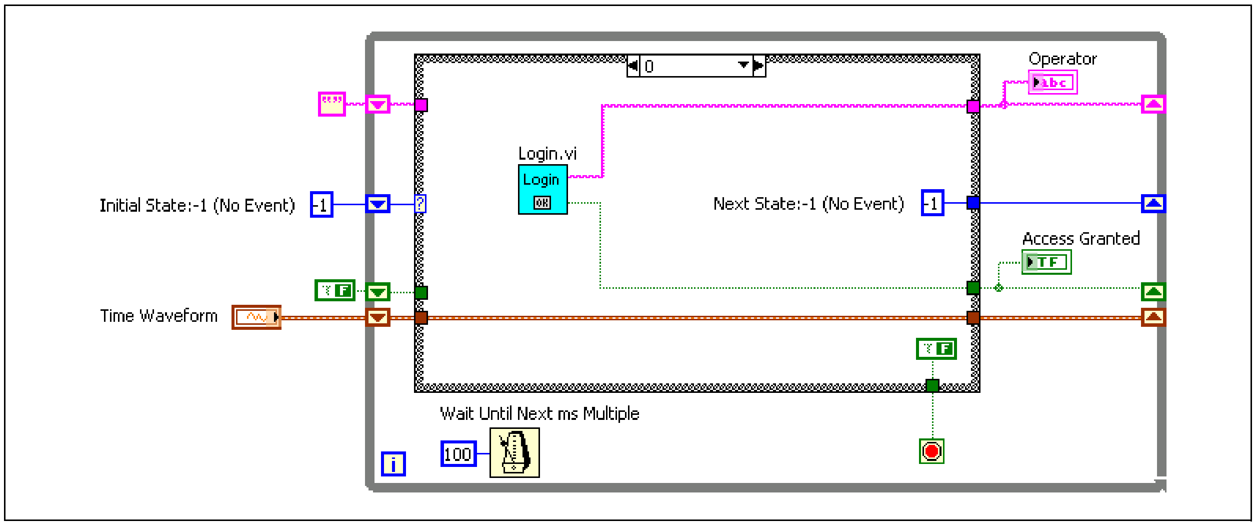
Note If you did not complete Exercise 3-4, use the State Machine with Analyze & Present Data VI located in the `C:\Solutions\LabVIEW Basics II` directory.

   a. Place a String indicator, located on the **Controls»Text Indicators** palette, on the front panel. Label it `Operator`.

   b. Place a Square LED, located on the **Controls»LEDs** palette, on the front panel and label it `Access Granted`.

Block Diagram

2. Open the block diagram and modify case 0.



a. Delete the One Button Dialog function.

b. Place the Login VI you built in Exercise 4-2 on the block diagram. Select **Functions»All Functions»Select a VI** and open the Login VI in the `C:\Exercises\LabVIEW Basics II` directory. The Login VI is called when the user presses the **Login** button on the front panel.

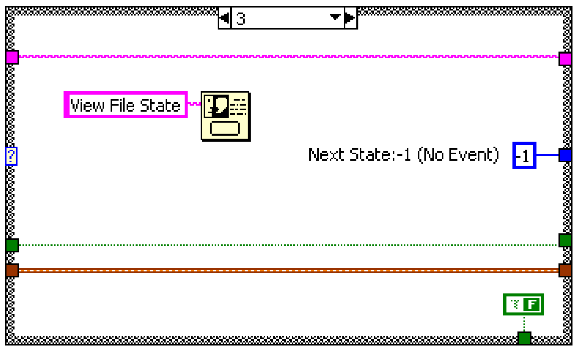
Note If you did not complete Exercise 4-2, use the Login VI located in the `C:\Solutions\LabVIEW Basics II` directory.

c. Place an Empty String constant, located on the **Functions»All Functions»String** palette, on the block diagram. This constant initializes the **Operator** indicator to an empty string when the VI starts.

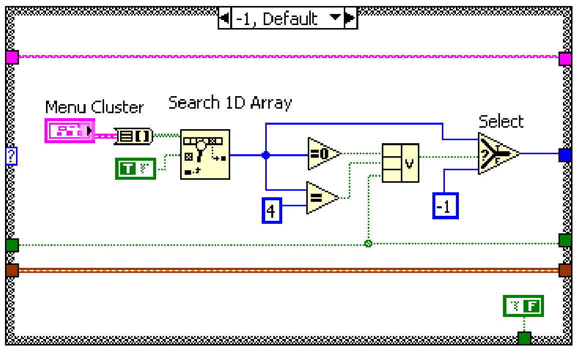
d. Place a False constant, located on the **Functions»Arithmetic & Comparison»Express Boolean** palette, on the block diagram. This constant initializes the **Access Granted** LED to off when the VI starts.

The VI needs the Operator and Access Granted information in subsequent loop iterations. Add two additional Shift Registers to store this data. By using Shift Registers, only Case 0, in which the Login VI runs, can change the Operator and Access Granted values.

Wire the Operator and Access Granted data straight through the other cases in the Case structure, as shown in the following block diagram.



3. Modify the –1 case as shown in the following block diagram.



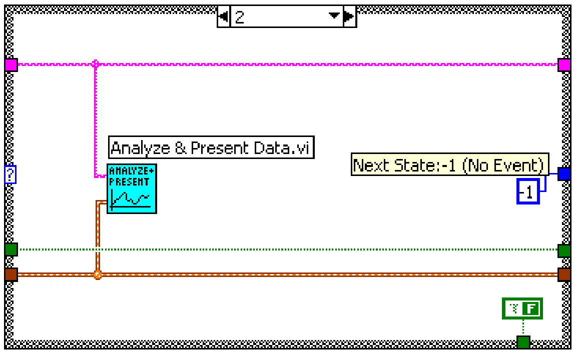
a. Place the Equal to 0? and Equal? functions, located on the **Functions»Arithmetic & Comparison»Express Comparison** palette, on the block diagram.

b. Place the Compound Arithmetic function, located on the **Functions»Arithmetic & Comparison»Express Numeric** palette, on the block diagram. Select the OR operation by right-clicking the function and selecting **Change Mode»OR** from the shortcut menu.

c. Place the Select function, located on the **Functions»Arithmetic & Comparison»Express Comparison** palette, on the block diagram. If the user has not logged in with a correct name and password, only the **Login** and **Stop** menu options execute. If the Login VI returns a value of FALSE for the access granted output, the **Operator** indicator shows an empty string.

The loop verifies that the value stored in the Boolean Shift Register, which is the Access Granted status, is TRUE, or if the user pressed the **Login** button (component 0 in the menu cluster) or **Stop** button (component 4) to determine which case to execute. If all of these conditions are FALSE, then Case –1 executes.

4. Wire the Operator value to the Acquire & Analyze VI in Case 2 as shown in the following block diagram.



5. Display the front panel and run the VI to test it. Use the LabVIEW debugging tools, such as execution highlighting, single-stepping, probes, and breakpoints, to determine the data flow of the VI.

6. Select **File»Save As** to save the VI as `State Machine with Login.vi` in the `C:\Exercises\LabVIEW Basics II` directory. Close the VI.

End of Exercise 4-3

C. Global Variables

Use global variables to access and pass data among multiple VIs that run simultaneously. Global variables are built-in LabVIEW objects. When you create a global variable, LabVIEW automatically creates a global VI, which has a front panel but no block diagram. Add controls and indicators to the front panel of the global VI to define the data types of the global variables it contains. In effect, this front panel is a container from which several VIs can access data.

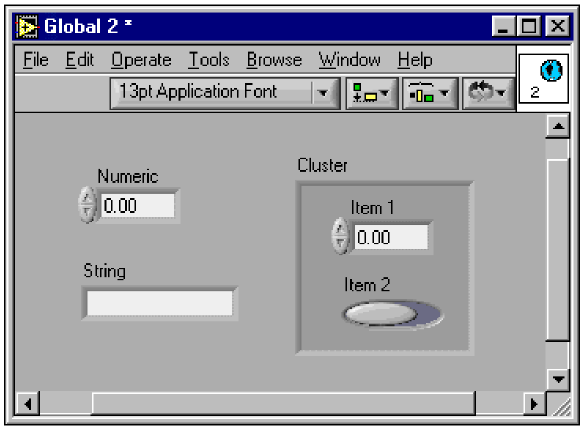
For example, suppose you have two VIs running simultaneously. Each VI contains a While Loop and writes data points to a waveform chart. The first VI contains a Boolean control to terminate both VIs. You must use a global variable to terminate both loops with a single Boolean control. If both loops were on a single block diagram within the same VI, you could use a local variable to terminate the loops.

Creating Global Variables

You can create several single global VIs, each with one front panel object, or you can create one global VI with multiple front panel objects. A global VI with multiple objects is more efficient because you can group related variables together.

Select **Functions»All Functions»Structures»Global Variable** to place a global variable, shown at left, on the block diagram. Double-click the global variable node to display the front panel of the global VI. Place controls and indicators on this front panel the same way you do on a standard front panel. LabVIEW uses owned labels to identify global variables, so label the front panel controls and indicators with descriptive owned labels.

The following example shows a global variable front panel with a numeric, a string, and a cluster containing a digital and a Boolean control. The toolbar does not show the **Run**, **Stop**, or related buttons as a normal front panel.



After you finish placing objects on the global VI front panel, save it and return to the block diagram of the original VI. You then must select which object in the global VI that you want to access. Right-click the global variable node and select a front panel object from the **Select Item** shortcut menu. The shortcut menu lists all the front panel objects that have owned labels. You also can use the Operating tool or Labeling tool to click the local variable node and select the front panel object from the shortcut menu.

If you want to use this global variable in other VIs, select **Functions»All Functions»Select a VI**. By default, the global variable is associated with the first front panel object with an owned label that you placed in the global VI. Right-click the global variable node you placed on the block diagram and select a front panel object from the **Select Item** shortcut menu to associate the global variable with the data from another front panel object.

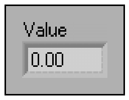
Exercise 4-4 Data to Global VI

Objective: To build a VI that writes data into a global variable.

Create a global variable you will use to send data to the VI in Exercise 4-5.

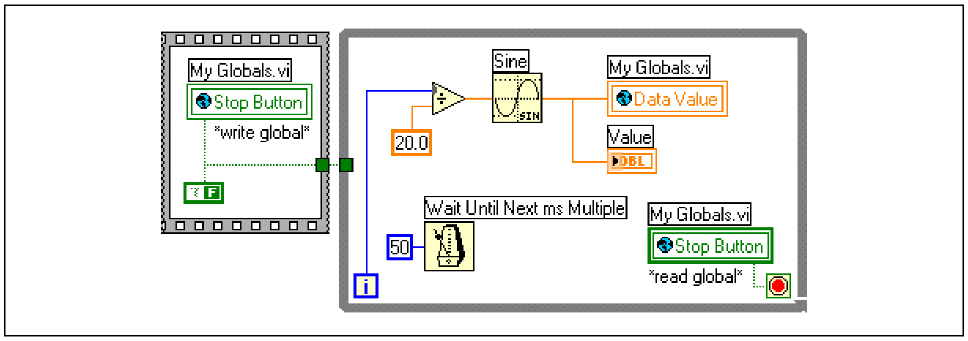
Front Panel

1. Open a blank VI and build the following front panel.



Block Diagram

2. Build the following block diagram. You will create the global variables in step 4.



a. Place a Stacked Sequence structure, located on the **Functions»All Functions»Structures** palette, on the block diagram. This structure initializes a global variable using artificial data dependency with the Boolean constant and the While Loop.

b. Place a While Loop, located on the **Functions»Execution Control** palette, on the block diagram. This structures the VI to continue running until a global Boolean sends a TRUE value.

c. Place a Wait Until Next ms Multiple function, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. Right-click the input terminal and select **Create»Constant** from the shortcut menu. Type `50` in the constant to write data to the global variable every 50 ms.

d. Place a Divide function, located on the **Functions»Arithmetic & Comparison»Express Numeric** palette, on the block diagram. In this exercise this function divides the iteration counter of the While Loop by 20. Right-click the bottom input and select **Create» Constant** from the shortcut menu. Type `20` in the constant.

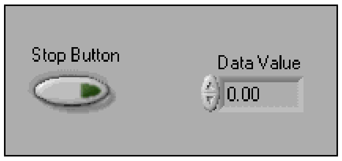
   e. Place a Sine function, located on the **Functions»Arithmetic & Comparison»Express Numeric»Express Trigonometric** palette, on the block diagram. This function accepts an input value in radians and outputs the sine of that value.

3. Select **File»Save As** to save the VI as `Data to Global.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

4. Create and add the My Globals VI global variables. These variables pass values between two concurrently running VIs. Complete the following steps to create and configure the global variables.

   a. Place a global variable node, located on the **Functions»All Functions»Structures** palette, on the block diagram of the Data to Global VI.

   b. Double-click the node to display the front panel for the global variable. Create the following global variable front panel using the owned labels shown. There is no block diagram associated with the global variable front panel.



   c. Save the global variable as `My Globals.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

   d. Close the global variable and return to the Data to Global VI block diagram.

   e. Right-click the global variable node and select **Visible Items»Label** from the shortcut menu.

   f. Click the global variable node with the Operating tool and select **Stop Button** from the **Select Item** shortcut menu. You can change the variable to be readable or writable by right-clicking it.

   g. You will need three copies of the `My Global.vi` variable. Use the Operating tool to select the item you need for each global variable.

5. Wire a FALSE constant to the Stop Button global variable inside the Sequence structure to initialize it. Wire the constant to the loop border to force the global to initialize before the loop begins executing. This prevents the While Loop from reading an uninitialized global variable, or one that has an unknown value.

6. Save the VI and keep it open so you can run it in the next exercise.

End of Exercise 4-4

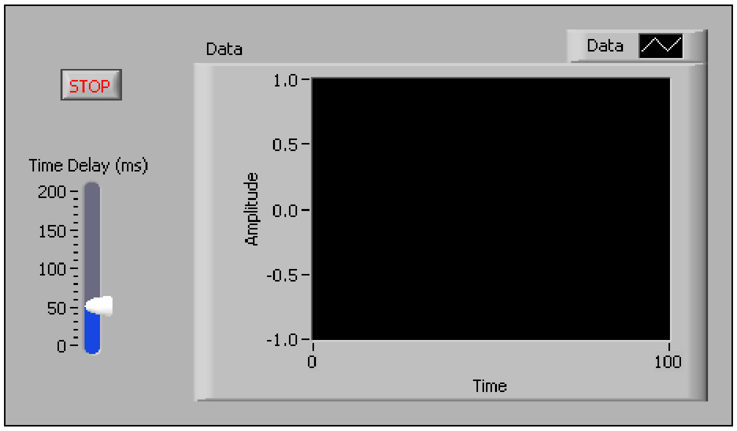
Exercise 4-5 Display Global Data VI

Objective: To build a VI that reads data from a global variable.

Build a VI that reads data from the global variable you created in the Exercise 4-4 and displays the data on a front panel chart.

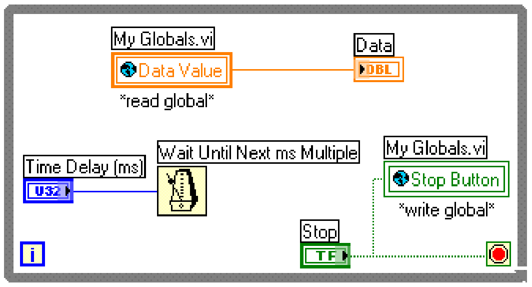
Front Panel

1. Open a new VI and build the following front panel using a vertical pointer slide control and waveform chart.



Block Diagram

2. Build the following block diagram.



a. Place a While Loop, located on the **Functions»Execution Control** palette, on the block diagram. The While Loop structures the VI to continue running until you press the Stop button.

b. Place a Wait Until Next ms Multiple function, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. This function sets the loop rate. Make sure the default is 20 iterations per second, or 50 ms.

3. Add the My Globals VI global variables to the block diagram. In this exercise, these variables pass values between two concurrently running VIs. These global variables were not created from a global variable node on this block diagram. Complete the following steps to create and configure them on this block diagram.

   a. Select **Functions»All Functions»Select a VI** and select `My Globals.vi`, located in the `C:\Exercises\LabVIEW Basics II` directory. The global variable object displayed in the Global Variable node is either a Stop button or Data Value, depending on the order in which they were placed on the front panel of the global variable.

   b. Copy the global variable so that you have two instances, one Stop Button global and one Data Value global. To access a different global variable object, click the node with the Operating tool and select the object you want from the **Select Item** shortcut menu.

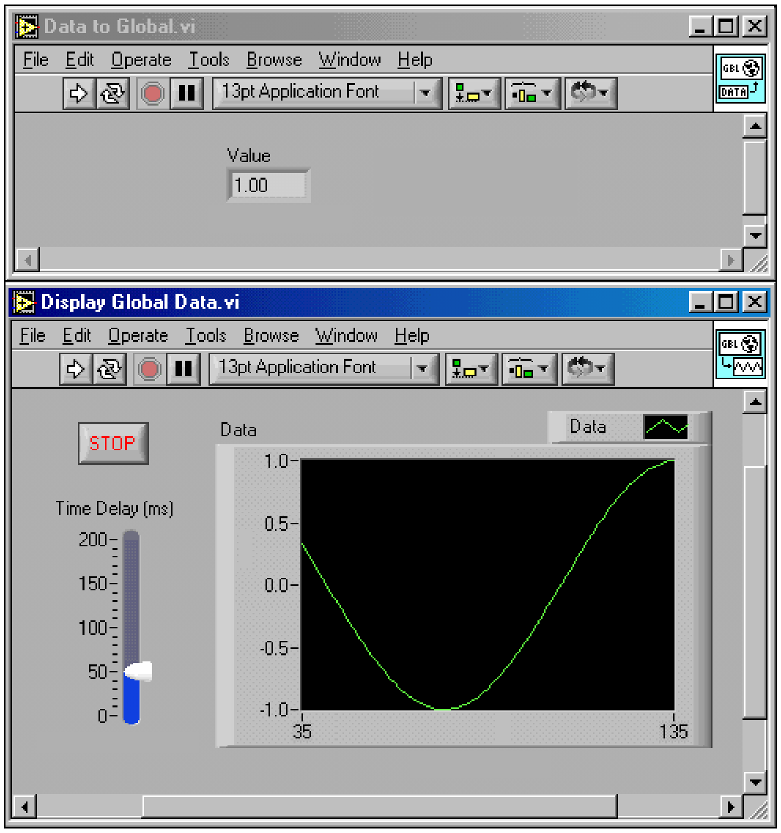
   c. Change the Data Value global to a read global by right-clicking the node and selecting **Change To Read** from the shortcut menu.

   The VI reads a value from the Data Value and passes the value to the waveform chart. It also writes the current value of the Stop button to the Stop Button global variable object each time through the loop. Using a global variable, the Boolean value is read in the Data to Global VI to control its While Loop.

4. Select **File»Save As** to save the VI as `Display Global Data.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

5. Display the front panel of the Data to Global VI and position the two front panels so both are visible.

6. Run the Data to Global VI. Switch back to the Display Global Data VI and run it. The waveform chart on the Display Global Data VI front panel displays the data. The Data to Global VI continually writes the value it calculates to the Data Value global variable object. The following example shows the Display Global Data VI reading the global variable object and updating the chart.



The Time Delay control determines how often the global variable is read. Notice how the Time Delay affects the values plotted on the waveform chart. If you set the Time Delay to 0, the same Data Value value is read from the global variable several times, appearing to decrease the frequency of the sine wave generated in the Data to Global VI. If you set the Time Delay to a value greater than 50 ms, the Display Global Data VI may never read some values generated in the Data to Global VI, and the frequency of the sine wave appears to increase. If you set the Time Delay to 50 ms, the same rate used in the While Loop in the Data to Global VI, the Display Global Data VI reads and displays each point of the Data Value global only once.

Note When using globals, if you are not careful, you may read values more than once, or you may not read them at all. If you must process every single update, take special care to ensure that a new value is not written to a global variable until the previous one has been read, and that after a global has been read, it is not read again until another value has been written to the global.

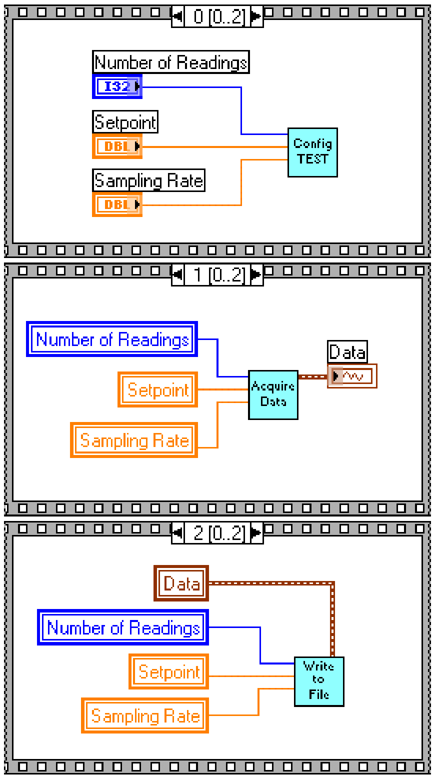
7. Click the **Stop** button on the Display Global Data VI front panel to stop the VI. Notice that both VIs stop. The VI continually writes the value of the **Stop** button to the Stop Button global variable object. That value is then read in the Data to Global VI and passed to the conditional terminal to control its loop as well. When you click the **Stop** button, a TRUE passes through the global variable to the Data to Global VI, where that TRUE value is read to stop that VI as well.
8. Close both VIs.

End of Exercise 4-5

D. Using Local and Global Variables Carefully

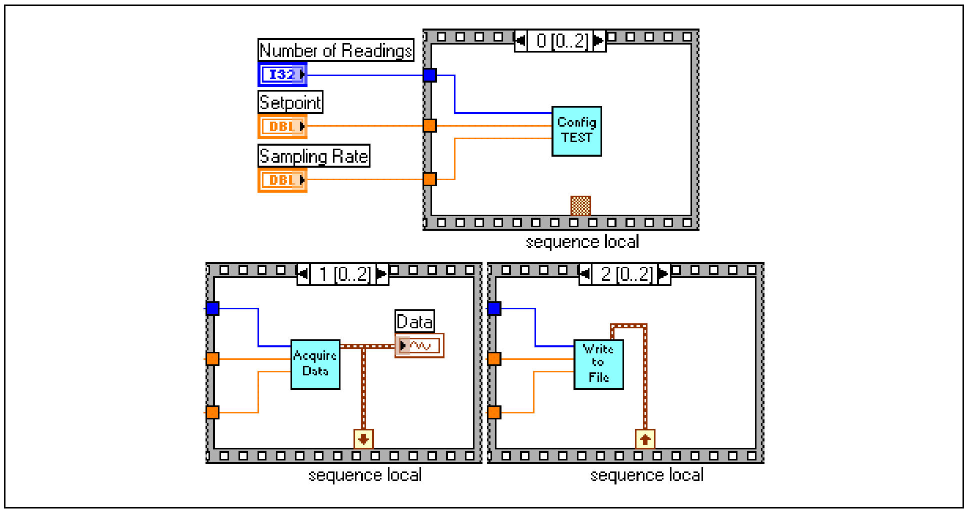
Local and global variables are inherently not part of the LabVIEW dataflow execution model. Block diagrams can become difficult to read when you use local and global variables, so you should use them carefully. Misusing local and global variables, such as using them instead of a connector pane or using them to access values in each frame of a Sequence structure, can lead to unexpected behavior in VIs. Overusing local and global variables, such as using them to avoid long wires across the block diagram or using them instead of data flow, slows performance.

The following three-frame Stacked Sequence structure performs the usual tasks for all VIs.

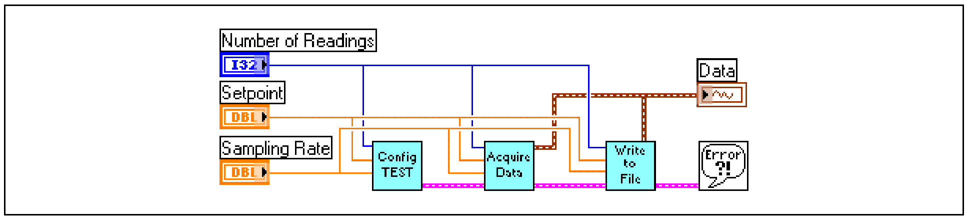


Frame 0 reads the controls on the panel for configuring the test system, Frame 1 uses the local variables for the controls to acquire data, and Frame 2 uses the locals to write the data to file. Although using a Stacked Sequence structure with a local variable does not cause errors, it is not the most efficient method of using dataflow programming. When you look at one of the frames, it is not obvious where the values for the locals are coming from and where they were last updated.

The following block diagram eliminates the use of local variables by putting the original control terminals outside the Stacked Sequence structure so that each frame of the sequence now has access to the values. The data are passed between frames of the sequence through a sequence local.



The following block diagram removes the Stacked Sequence structure and uses data flow to define how the program works.



The error clusters define the execution order of the subVIs and also maintain the error conditions which are checked at the end with the error handler VI. This is the most efficient way to build a program and manage data in LabVIEW. Not only is the block diagram smaller and easier to read, but passing data through wires is the most efficient method for memory use.

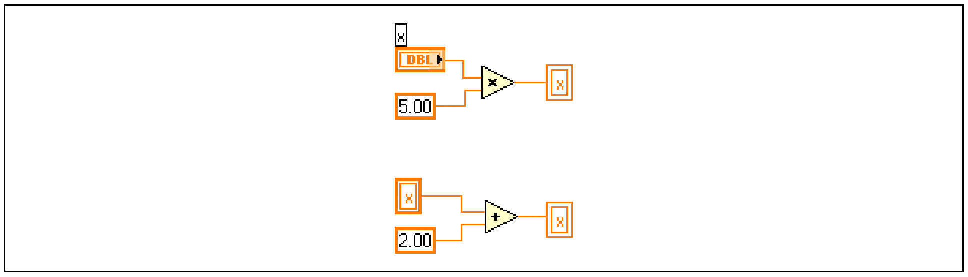
Initializing Local and Global Variables

Verify that the local and global variables contain known data values before the VI runs. Otherwise, the variables might contain data that cause the VI to behave incorrectly.

If you do not initialize the variable before the VI reads the variable for the first time, the variable contains the default value of the associated front panel object.

Race Conditions

Because VIs follow a dataflow execution model, LabVIEW local and global variables do not behave like local and global variables in text-based programming languages. A race condition occurs when two or more pieces of code that execute in parallel change the value of the same shared resource, typically a local or global variable. The following example is a race condition.



The output of this VI depends on the order in which the operations run. Because there is no data dependency between the two operations, there is no way to determine which runs first. To avoid race conditions, do not write to the same variable you read from. The best use for variables is to write to a variable from one location only. It can then be read from many locations.

If you use global variables in VIs that execute in parallel, you can use an additional Boolean global variable to indicate when global variable data changes. Other VIs can monitor this Boolean global variable to determine if the data changed and to read the new value.

Summary, Tips, and Tricks

- You can use global and local variables to access a given set of values throughout your LabVIEW application. These variables pass information among places in your application that cannot be connected by a wire.
- Local variables access front panel objects of the VI in which you placed the local variable.
- When you write to a local variable, you update its corresponding front panel control or indicator.
- When you read from a local variable, you read the current value of its corresponding front panel control or indicator.
- Global variables are built-in LabVIEW objects that pass data among VIs. They have front panels in which they store their data.
- Always write a value to a global variable before reading from it, so that it has a known initial value when you access it.
- Write to local and global variables at locations separate from where you read them to avoid race conditions.
- Use local and global variables only when necessary. Overuse can slow execution and cause inefficient memory usage in your application.
- Because local and global variables do not use data flow, if you use them too frequently, they can make your block diagrams difficult for others to understand. Use locals and globals wisely.

Additional Exercises

4-6 Open the In Range VI located in the `C:\Exercises\LabVIEW Basics II` directory. Examine the block diagram. This simple VI generates five random numbers and turns on an LED if the last number generated is between 0.1 and 0.9. Run the VI several times until the LED turns on. Notice that if the LED turns on during one run of the VI, it remains on during the second run until the For Loop completes and the last number is passed to the In Range? function. Modify the VI using a local variable so that the LED is turned off when the VI starts execution. Save the VI after you finish.

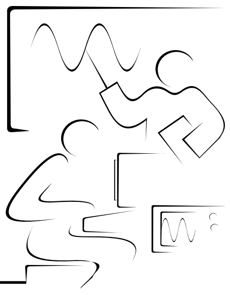
4-7 The transfer of data through a global variable is not a synchronized process. If you try to read data from a global variable too quickly or slowly, you may read copies of the same value or skip data points entirely. This was shown in Exercise 4-5, where you could adjust the time delay between reads of the Data Value to read the global faster or slower than the data was actually available.

Modify the Data to Global VI and Display Global Data VI, from Exercises 4-4 and 4-5, so they use a handshaking protocol to make sure that no data points are skipped or read multiple times. Set up an additional Boolean in `My Global.vi` named `Handshake` that indicates when VIs are ready to send or receive data. In the Data to Global VI, set the Handshake global Boolean to FALSE before it passes a new data point to the Data Value numeric. In the Display Global Data VI, set the Handshake Boolean to TRUE before it attempts to read the numeric data. In both VIs, set the Handshake Boolean so the other VI knows when to access the variable.

Save the new global Boolean as `Handshake Global.vi`, and the other two VIs as `Data to Handshake Global.vi` and `Display Handshake Global.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

Notes

Lesson 5
Advanced File I/O Techniques



In the *LabVIEW Basics I: Introduction* course, you learned the fundamentals of file I/O. This lesson describes advanced file I/O techniques. For example, one data format may have advantages over a different data format for your application. In addition, you may need to optimize the speed of your file I/O operations. Both the data format and the structure of your file I/O routines affect speed, so careful planning may be required. This lesson provides the in-depth understanding required for effective, flexible file I/O operations.

You Will Learn:

A. How to work with byte stream files

B. How to create and work with datalog files

C. About streaming data to disk

D. About the advantages and disadvantages of text, binary, and datalog files

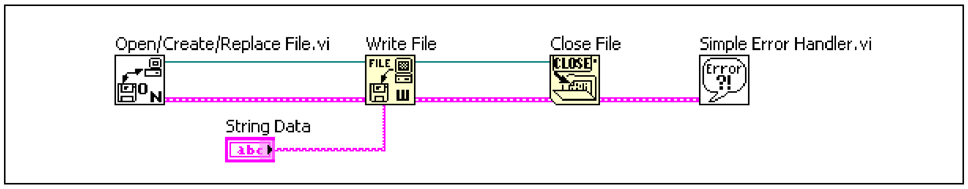
A. Working with Byte Stream Files

Byte Stream files include both text and binary files. The same LabVIEW functions are used to manipulate both text and binary byte stream-type files. However, binary files are machine readable only, unlike text files, which are human readable. Also, because you cannot rely on special characters such as the <Tab> and <Return> keys, you must know the structure of the data stored in the file before reading it. Without this knowledge, you cannot successfully interpret the data stored in the binary file.

Frequently Used File I/O Functions for Byte Stream Files

The following file I/O functions are located on the **Functions»All Functions»File I/O** and **Functions»All Functions»File I/O»Advanced File Functions** palettes:

- The Open/Create/Replace File function opens an existing file, creates a new file, or replaces an existing file.
- The Write File function writes data to an open file. You can use the Write File function to create both types of byte stream files—text and binary. Creating a text file is as simple as sending a string containing characters to the Write File function.



- The Read File function reads data from an open file.
- The Close File function closes an open file.

When to Use Text Files

Use text format files for your data to make it available to other users or applications, if disk space and file I/O speed are not crucial, if you do not need to perform random access reads or writes, and if numeric precision is not important.

Text files are the easiest format to use and to share. Almost any computer can read from or write to a text file. A variety of text-based programs can read text-based files. Most instrument control applications use text strings.

Store data in text files when you want to access it from another application, such as a word processing or spreadsheet application. To store data in text format, use the String functions located on the **Functions»All Functions» String** palette to convert all data to text strings. Text files can contain information of different data types.

Text files typically take up more memory than binary and datalog files if the data is not originally in text form, such as graph or chart data, because the ASCII representation of data usually is larger than the data itself. For example, you can store the number –123.4567 in 4 bytes as a single-precision floating-point number. However, its ASCII representation takes 9 bytes, one for each character.

You might lose precision if you store numeric data in text files. Computers store numeric data as binary data, and typically you write numeric data to a text file in decimal notation. A loss of precision might occur when you read the data from the text file. Loss of precision is not an issue with binary files.

The Write File function can write both text and binary files. The data terminal of the Write File function is polymorphic, which means that it adapts to the kind of data you wire into it. Thus, you could create a binary file by wiring binary data to the Write File function in the previous example. However, header information is vital for interpreting binary data.

For example, if you wire a 2D array of numbers into the **data** terminal, the Write File function places a byte-for-byte copy of the input data into the file. It does not convert the numbers to text, however it can place information about the number of rows and columns in the array into the file. If the original data consists of two rows and four columns of double-precision floating-point numbers, you can reconstruct the 2D array if you have the information contained in the header. If the file contains 64 bytes of data, and knowing that double-precision floating-point numbers use eight bytes of memory each, you can figure out that the array contained eight values (64/8 = 8). However, without the header information the original data might have been stored in a one-dimensional array of eight elements, a 2D array with one row and eight columns, or a table with four columns and two rows. A file header provides the file format information needed to reconstruct the original array.

When to Use Binary Files

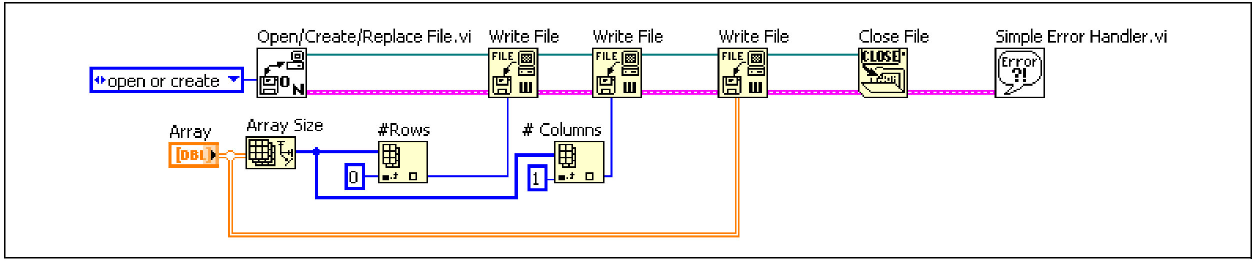
A binary file uses a fixed number of storage bytes on disk. For example, storing any number from 0 to 4 billion in binary format, such as 1, 1,000, or 1,000,000, takes up 4 bytes for each number.

Use binary files to save numeric data and to access specific numbers from a file or randomly access numbers from a file. Binary files are machine readable only, unlike text files, which are human readable. Binary files are the most compact and fastest format for storing data. You can use multiple data types in binary files, but it is uncommon.

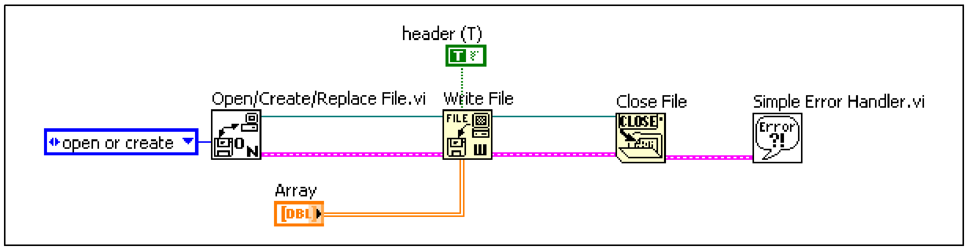
Binary files are more efficient because they use less disk space and because you do not need to convert data to and from a text representation when you store and retrieve data. A binary file can represent 256 values in 1 byte of disk space. Often, binary files contain a byte-for-byte image of the data as it was stored in memory, except for cases like extended and complex numerics. When the file contains a byte-for-byte image of the data as it was stored in memory, reading the file is faster because conversion is not necessary.

Creating Header Information for a Binary File

When you create binary files, designing an appropriate header for the file is often the single most important task. You can create the header information by explicitly generating a header or by using the **header** input of the Write File function. The following example shows how to explicitly generate a header that contains the number of rows and columns of data.

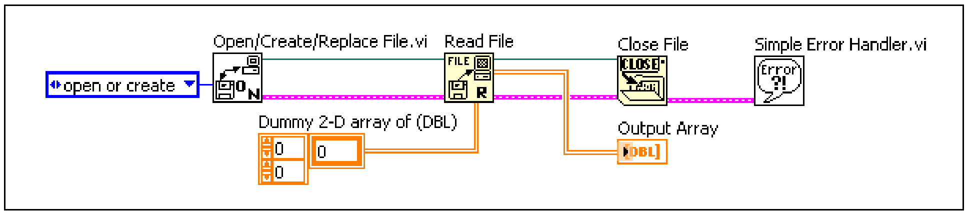


You also can generate the same file using the **header** input of the Write File function. If **header** is TRUE, it writes the same data to the file as if you had manually written a header. The following example shows how to use the Write File function to create a binary file with a header.



Using the previous example, if you again wire a 2D array of double-precision numbers with two rows and four columns, the file would contain 64 bytes of data and two additional long integers, 4 bytes each, as header information. The first 4 bytes contain the number of rows in the array; the second 4 bytes contain the number of columns. Therefore, the entire file would be 64 + 4 + 4 = 72 bytes in length.

As shown in the following example, you can read the binary file from the previous examples by using the Read File function.

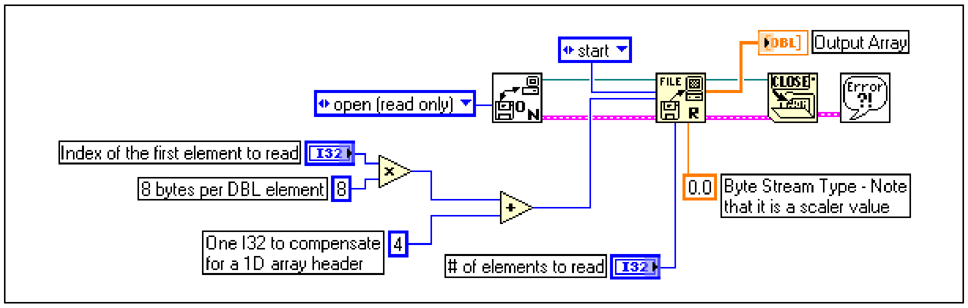


The **byte stream type** input of the Read File function has a 2D array of double-precision numbers wired to it. The Read File function uses only the data type of this input to read the data in the file, assuming that it has the correct header information for that data type.

Random Access in Byte Stream Files

It is difficult to randomly access numeric data in text files. Although each character in a string takes up exactly 1 byte of space, the space required to express a number as text typically is not fixed. To find the ninth number in a text file, LabVIEW must first read and convert the preceding eight numbers.

Use binary files to randomly access numbers from a file. Often, binary files contain a byte-for-byte image of the data as it was stored in memory, except for cases like extended and complex numerics. When the file contains a byte-for-byte image of the data as it was stored in memory, reading the file is faster because conversion is not necessary. If you know that a file stores double-precision numbers, which use 8 bytes per number, you can read an arbitrary group of elements from the array, as shown in the following example.


start
Output Array
open (read only)
Index of the first element to read
8 bytes per DBL element
One I32 to compensate for a 1D array header
# of elements to read
Byte Stream Type - Note that it is a scaler value
0.0

In the previous example, the **pos mode** input of the Read File function is set to `start`, which means that the function begins reading data at **pos offset** bytes from the beginning of the file. The first byte in the file has an offset of zero. So, the location of the n^{th} element in an array of single-precision floating-point numbers stored in this file is $8 \times n$ bytes from the beginning of the file. A double-precision floating-point constant, wired to the **byte**

stream type input, tells the Read File function to read double-precision floating-point values from the file. The **# of elements to read** control, connected to the **count** input of the Read File function, tells the function how many double-precision elements to read from the file.

The following points are important to remember about random access operations:

- When performing binary file I/O, remember that values for the **pos offset** terminal are measured in bytes.
- The **count** input in the Read File function controls how many bytes of information are read from the file when the **byte stream type** input is unwired. The data read from the file is returned in a string.
- If you wire **count**, but you do not wire **byte stream type**, **data** is a string.
- If you wire **count** and **byte stream type**, the function returns an array containing **count** elements of the same data type as **byte stream type**.

Exercise 5-1 Binary File Writer VI

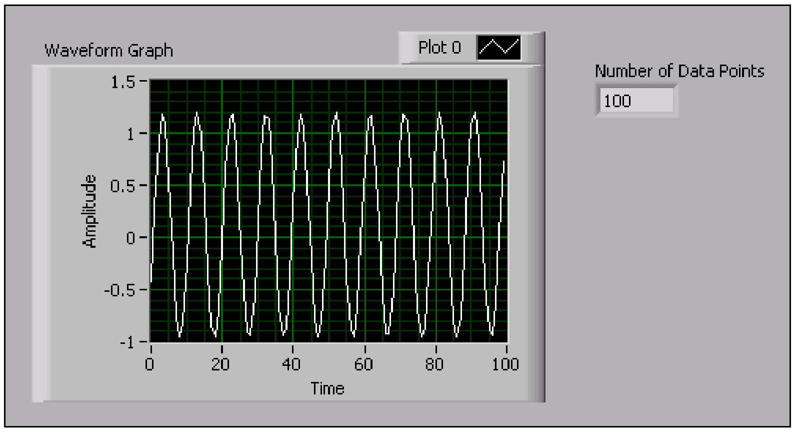
Objective: To build a VI that writes data to a binary file with a simple data formatting scheme.

Build a VI that saves data to a binary file using a simple formatting scheme in which the header for the file is a 32-bit signed integer (I32) containing the number of data points in the file. In Exercise 5-2, you build a VI that reads the binary file you create with this VI.

Note If you do not have a DAQ device, use the Simulate Signal Express VI, located on the **Functions»Input** palette, instead of the DAQ Assistant Express VI in step 2a. Use the default settings of the Simulate Signal Express VI to create data to write to a file.

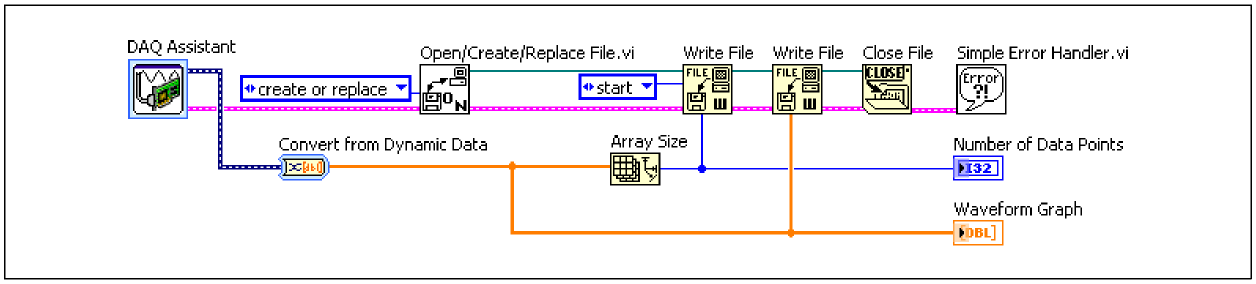
Front Panel

1. Open a new VI and build the following front panel. Right-click the **Number of Data Points** indicator and select **Representation»I32** from the shortcut menu.



Block Diagram

2. Open and build the following block diagram.



a. Place the DAQ Assistant Express VI, located on the **Functions» Input** palette, on the block diagram to launch MAX.

Complete the following steps to configure MAX.

(1) Select **Analog Input»Voltage** for the measurement to make.

(2) Select ***`Your DAQ Device`*****»ai1** for the physical channel.

(3) The **Analog Voltage Task** configuration dialog box appears. Click the **OK** button to accept the default settings and close the dialog box.

(4) Right-click the DAQ Assistant Express VI and select **View As Icon** from the shortcut menu to conserve space on the block diagram.

b. Place the Convert from Dynamic Data Express VI, located on the **Functions»Signal Manipulation** palette, on the block diagram. This Express VI converts the dynamic data into an array of doubles. This enables the Write File function to store the data as an array of doubles, and not as dynamic data.

c. Place the Open/Create/Replace File VI, located on the **Functions» All Functions»File I/O** palette, on the block diagram. This VI creates or replaces a file.

d. Place the Write File function, located on the **Functions»All Functions»File I/O** palette, on the block diagram. You use two of these functions. The first instance writes the binary file's header information, which is a four-byte integer containing the number of values written to the file. The second instance writes the array of data to the file.

e. Place the Close File function, located on the **Functions»All Functions»File I/O** palette, on the block diagram. In this exercise, this function closes the binary file after data has been written to it.

f. Place the Simple Error Handler VI, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. In the event of an error, this VI displays a dialog box with information about the error and where it occurred.

g. To create this constant, right-click the **pos mode** input of the Write File function and select **Create»Constant**. Set the position mode to `start` to ensure that new data are written relative to the beginning of the file.

h. To create this constant, right-click the **function** input of the Open/Create/Replace File VI and select **Create»Constant**. By selecting **create or replace**, you allow the user to create a new file or overwrite an existing file.

i. Place the Array Size function, located on the **Functions»All Functions»Array** palette, on the block diagram. In this exercise, this function returns the number of elements in the 1D array of data to be written to the file.

3. Select **File»Save As** to save the VI as `Binary File Writer.vi` in the `C:\Exercises\LabVIEW Basics II` directory.
4. On the DAQ Signal Accessory, wire the Sine Wave Function Generator to Analog Input 1.
5. Display the front panel and run the VI. Save the data as `data.bin` in the `C:\Exercises\LabVIEW Basics II` directory.
6. Close the VI.

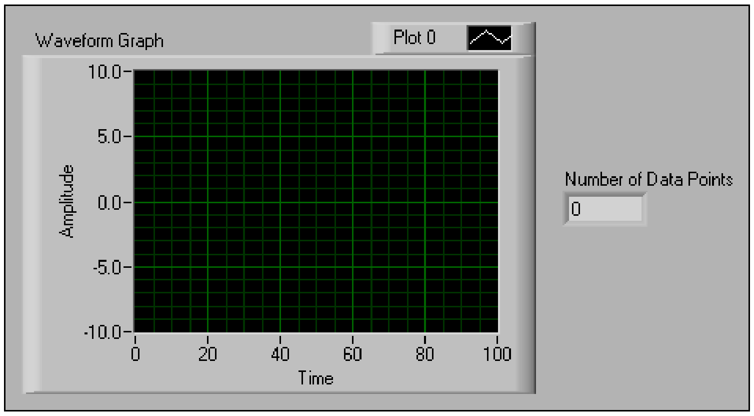
End of Exercise 5-1

Exercise 5-2 Binary File Reader VI

Objective: To build a VI that reads the binary file created in the previous exercise.

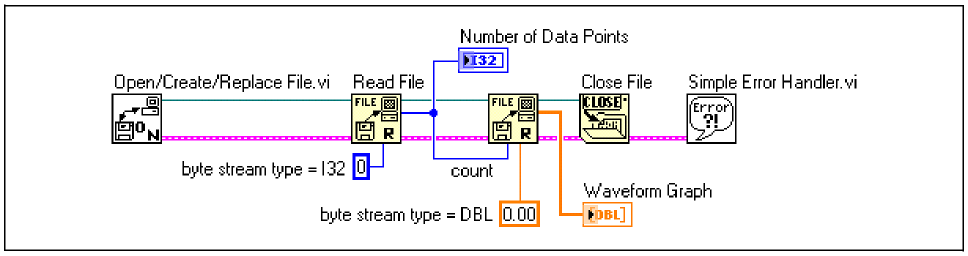
Front Panel

1. Open a new VI and build the following front panel. Right-click the **Number of Data Points** indicator and select **Representation»I32** from the shortcut menu.



Block Diagram

2. Open and build the following block diagram.



a. Place the Open/Create/Replace File VI, located on the **Functions»All Functions»File I/O** palette, on the block diagram. This VI opens a file.

b. Place the Read File function, located on the **Functions»All Functions»File I/O** palette, on the block diagram. You use two of these functions in this exercise. The first instance reads the binary file's header information, which is a four-byte integer containing the number of elements in the array stored in the file. The second instance reads the array of data from the file.

c. Place the Close File function, located on the **Functions»All Functions»File I/O** palette, on the block diagram. In this exercise, this function closes the binary file after data has been read from it.

d. Place the Simple Error Handler VI, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. In the event of an error, this VI displays a dialog box with information about the error and where it occurred.

e. Place the Numeric constant, located on the **Functions»Arithmetic & Comparison»Express Numeric** palette, on the block diagram. Set the data representation of the constant to I32 so the Read File function knows what data type to expect. I32 is the default data type for numeric constants.

f. Place the Numeric constant, located on the **Functions»Arithmetic & Comparison»Express Numeric** palette, on the block diagram. Create a double-precision constant so the Read File function knows what data type to expect. Because the default data type for numeric constants is I32, right-click the constant and select **Representation» Double Precision**.

3. Display the front panel. Select **File»Save As** to save the VI as `Binary File Reader.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

4. Run the VI. When prompted, open the `data.bin` file, located in the `C:\Exercises\LabVIEW Basics II` directory.

   After the file opens, the Read File function uses the **byte stream type** input to read the first four bytes from the file. The function displays the number in the Number of Data Points indicator, which shows how many numbers were stored in the file. Recall that this header was created by the Write File function in the Binary File Writer VI in Exercise 5-1.

   The second Read File function reads the array of data points from the file using the **byte stream type** input, which has a double-precision floating-point value wired to it. The **count** input specifies how many values should be read from the file.

5. Close the VI.

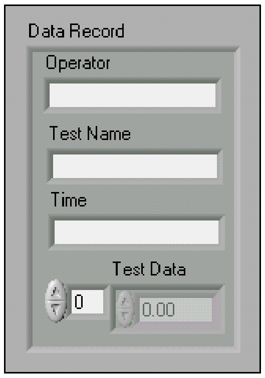
End of Exercise 5-2

B. LabVIEW Datalog Files

If your data has a mixture of data types, formatting it into text or binary strings for storage can be tedious or inefficient. LabVIEW datalog files use a data storage format for saving records of data of an arbitrary data type. These records consist of clusters of data. For instance, a cluster could contain a device ID, time stamp, and an array of acquired data. Each time the data is acquired, the new ID, time stamp, and array becomes one record that is saved to a disk file.

Use datalog files to access and manipulate data only in LabVIEW and to store complex data structures quickly and easily. Datalog files can only be created and read by LabVIEW.

A datalog file stores data as a sequence of identically structured records, similar to a spreadsheet, where each row represents a record. Each record in a datalog file must have the same data types associated with it. LabVIEW writes each record to the file as a cluster containing the data to store. However, the components of a datalog record can be any data type, which you determine when you create the file. In the following example, each record consists of a cluster containing the name of the test operator, information about the test, a time stamp, and an array of numeric values for the actual test data.



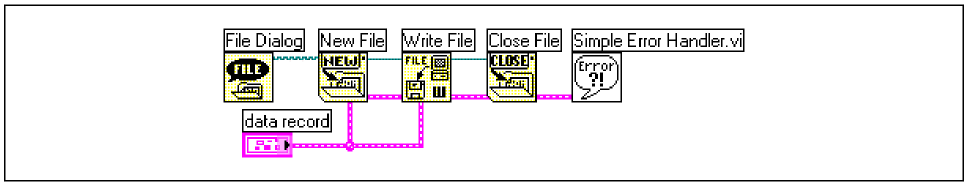
Frequently Used File I/O Functions for Datalog Files

Datalog file I/O uses three separate file I/O functions, instead of the Open/Create/Replace function. These file I/O functions are located on the **Functions»All Functions»File I/O»Advanced File Functions** palette.

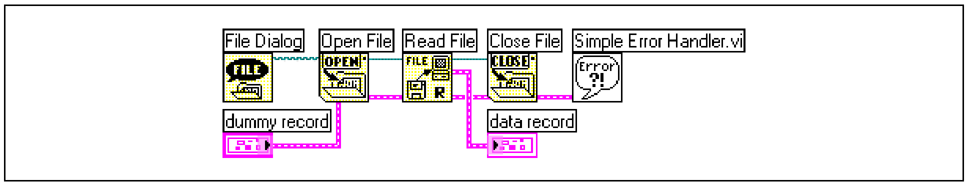
- The File Dialog function displays a dialog box for file selection. You can use this dialog box to select existing files or directories or to select a location and name for a new file or directory.
- The Open File function opens an existing file for reading or writing. You cannot use this function to create or replace files.

- The New File function creates a new file and opens it for reading or writing.

To create a new datalog file, wire a cluster matching the data record cluster to the **datalog type** terminal of the New File function. This cluster specifies how the data is written to the file. Then wire the actual data record to the **data** terminal of the Write File function. The **datalog type** cluster wired to New File can be, but need not be, an actual data record—you need it only to establish the format of the cluster to be stored in the file. The following example shows how to create a new datalog file.



To read the record of information from the datalog file, you must wire an input to the **datalog type** of the Open File function that exactly matches the data type of the records stored in the file. The following example shows how to open and read an existing datalog file.



Note The cluster wired to the **datalog type** input of the Open File function must be identical to the cluster used to create the file and write data to it, including numeric data types and cluster order.

When the **count** input of the Read File function remains unwired, the function reads a single record from the datalog file. If you wire an input to the **count** terminal, Read File returns an array of records.

Exercise 5-3   Save Data to File VI

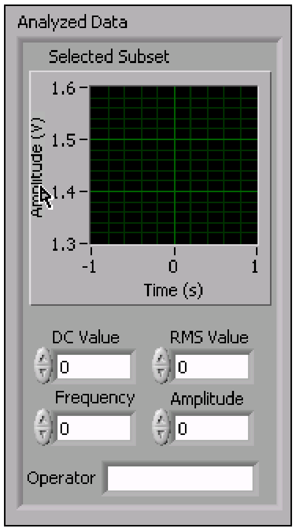
Objective: To complete a VI that saves data in a datalog file.

Note Completing this VI enables the Log Results to File option in the Analyze & Present Data VI from Exercise 3-3.

In the final application you are developing, you need to save a mixed data set of simple numeric values, arrays of numeric values, and string data to disk. In addition, this data is used only in LabVIEW. Because of these requirements, you will use datalog files to save the application's data subsets to disk.

Front Panel

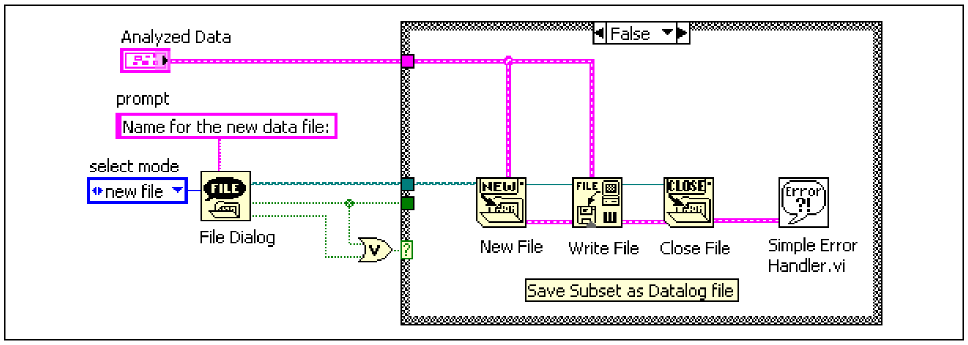
1. Open the Save Data to File VI located in the `C:\Exercises\LabVIEW Basics II` directory.
2. Place the Type Definition control that you created in Exercise 3-6 on the front panel. Select **Controls»All Controls»Select a Control** and navigate to `C:\Exercises\LabVIEW Basics II\Extracted Data.ctl` to place the control on the front panel.



Note You might need to right-click the cluster and select **Change to Control** from the shortcut menu to make it a control.

Block Diagram

3. Open and build the following block diagram.



a. Delete the Boolean constant labeled `Delete Me`.

b. Place the File Dialog function, located on the **Functions»All Functions»File I/O»Advanced File Functions** palette, on the block diagram. This function prompts for the name of the new file. Connect the output of the **exists** and **cancelled** terminals to the inputs of the Or function. Wire the **exists** value to the Case structure as shown in the previous block diagram.

new file

c. Right-click the **select mode** input of the File Dialog function and select **Create»Constant** from the shortcut menu. Click the constant with the Operating tool and set it to the value **new file**.

Name for the new data file:

d. Right-click the **prompt** input of the File Dialog function and select **Create»Constant** from the shortcut menu. This string displays a prompt message in the file dialog box. Type the text shown at left into the constant.

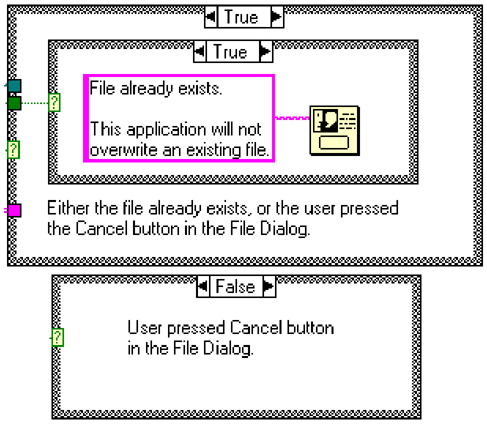
e. Place the New File function, located on the **Functions»All Functions»File I/O»Advanced File Functions** palette, on the block diagram. This function creates the new file. The **datalog type** input is located in the middle of this function. Move the cursor over this function and refer to the **Context Help** window for more information about where the input terminals are located.

f. Place the Write File function, located on the **Functions»All Functions»File I/O** palette, on the block diagram. This function writes a record to the datalog file.

g. Place the Close File function, located on the **Functions»All Functions»File I/O** palette, on the block diagram. In this exercise, this function closes the file after the data is written to it.

h. Place the Simple Error Handler VI, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. In the event of an error, this VI displays a dialog box with information about the error and where it occurred.

4. Examine the following Case structures already built for the block diagram.


True
True
File already exists.
This application will not overwrite an existing file.
Either the file already exists, or the user pressed the Cancel button in the File Dialog.
False
User pressed Cancel button in the File Dialog.

   The outside Case structure writes the data to a LabVIEW datalog file. Bundling the Employee Name and Data Cluster together into the Analyzed Data cluster provides the data type for the datalog file.

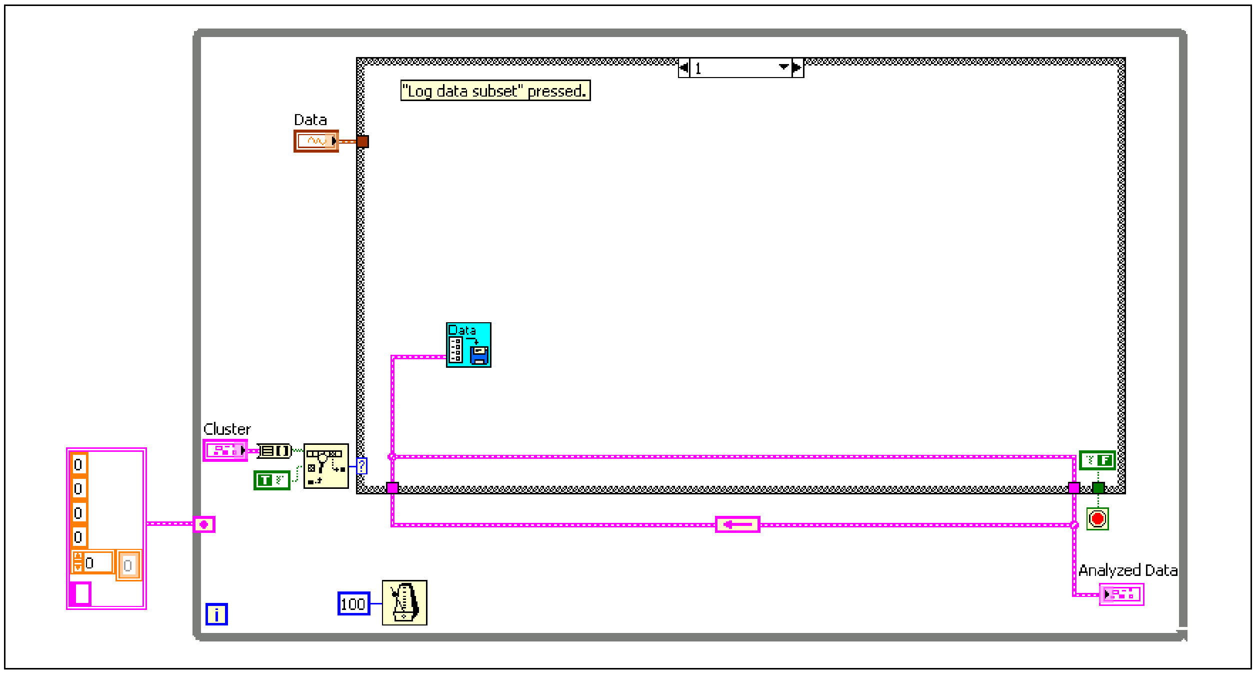
5. Display the front panel. Connect the Analyzed Data cluster to the connector pane.

   a. Right-click the VI icon in the upper right corner of the front panel and select **Show Connector** from the shortcut menu.

   b. Click the Analyzed Data cluster with the Wiring tool, then click a terminal in the connector pane to associate the cluster with the terminal.

6. Save and close the VI.

7. Open the Analyze & Present Data VI from Exercise 3-3. Add the Save Data to File subVI to case 1 as shown in the following figure.



8. Save the VI.
9. Run the Analyze & Present Data VI. Select a subset of data and click the **Analyze Selected Subset** button. Click the **Log Results to File** button. A dialog box prompts you to name the data file to save. Type the name `subset1.dat` to save the data set as a datalog file in the `C:\Exercises\LabVIEW Basics II` directory.
10. Click the **Return** button and close the Analyze & Present Data VI when you are finished.

End of Exercise 5-3

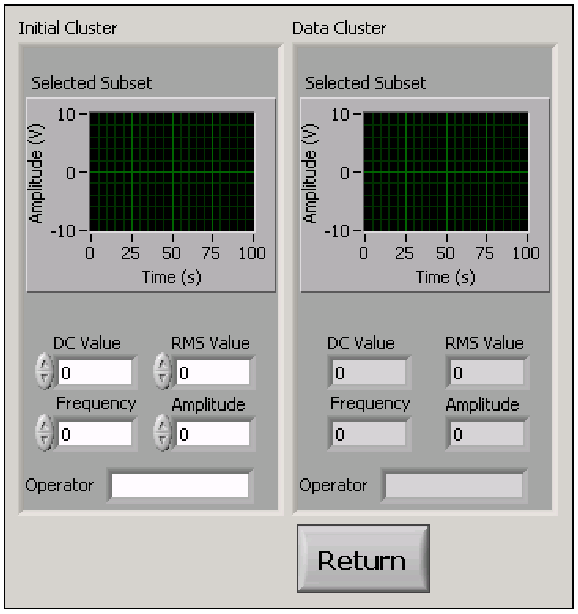
Exercise 5-4 View Analysis File VI

Objective: To study a VI that reads data files created by the Save Data to File VI from Exercise 5-3.

This VI reads and displays the data stored by the Save Data to File VI.

Front Panel

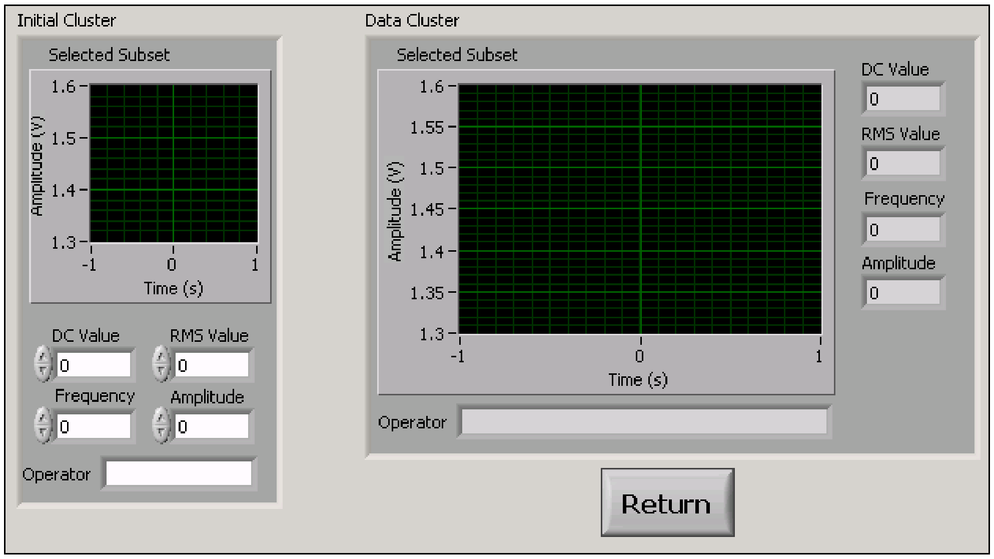
1. Open the View Analysis File VI located in the `C:\Exercises\LabVIEW Basics II` directory.



   a. Place two type definition controls that you created in Exercise 3-6 on the front panel. Select **Controls»All Controls»Select a Control** and navigate to `C:\Exercises\LabVIEW Basics II\Extracted Data.ctl` to place the type definition controls on the front panel.

   b. Label the clusters **Initial Cluster** and **Data Cluster**. Make sure that **Initial Cluster** is a control and **Data Cluster** is an indicator.

2. Resize the **Data Cluster** and reorganize the data items in the cluster similar to the following front panel.

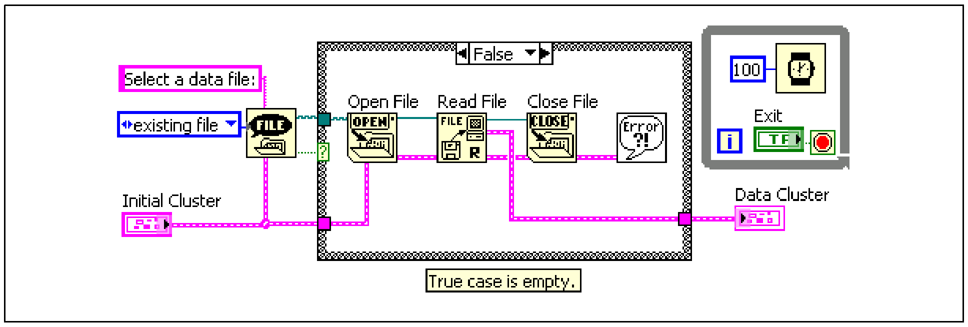


Note If the Extracted Data control was a strict type definition you would be unable to move the data items in **Data Cluster**.

3. Resize the window to only show **Data Cluster** and the **Return** Boolean button. **Initial Cluster** is only used to initialize the data and does not need to be visible to the user.

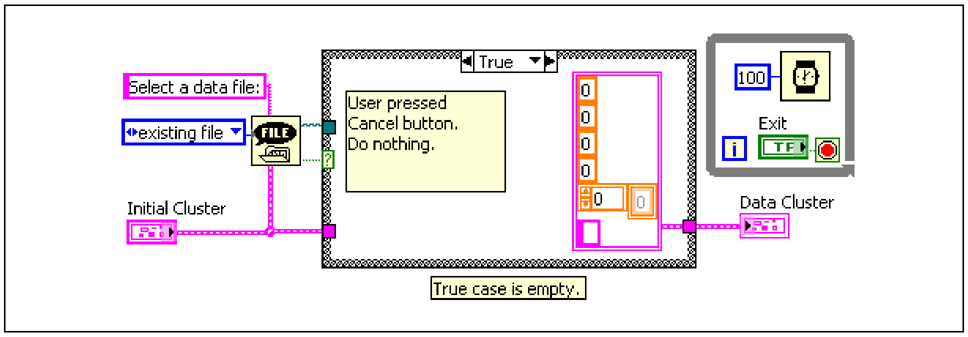
Block Diagram

4. Complete the following block diagram.



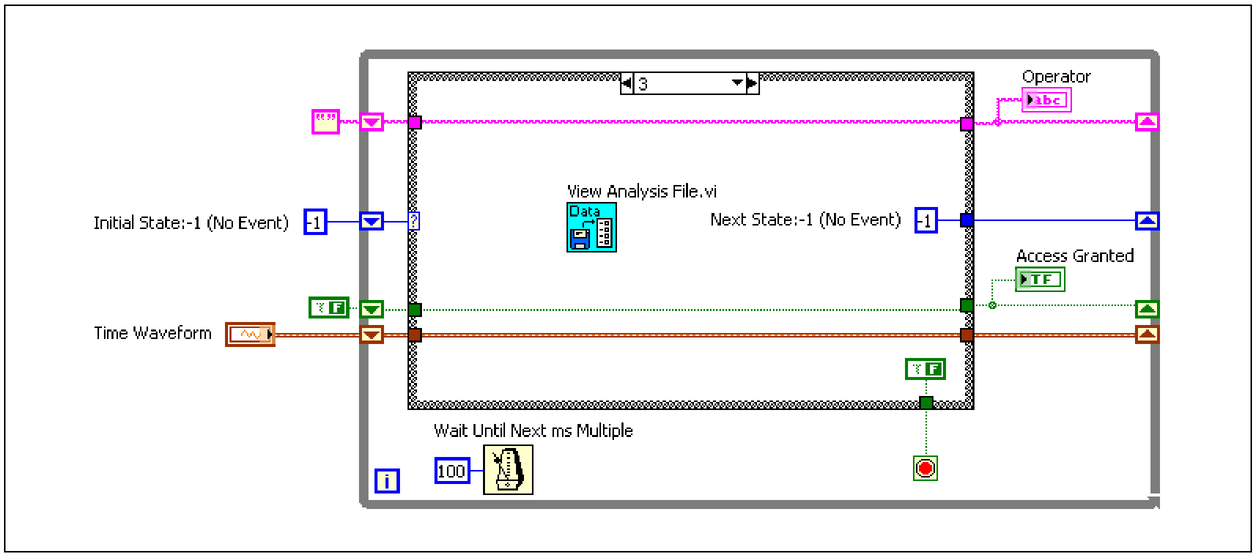
a. Wire **Initial Cluster** to the **datalog type** input of the File Dialog function and Open File function.

b. Right-click **Initial Cluster** and select **Create»Constant** from the shortcut menu. Place the constant in the True case and wire it to the output tunnel of the True case, as shown in the following figure.



5. Save the VI.
6. Display the front panel and run the VI. Examine the behavior of the VI. If the user cancels the file dialog box, nothing happens.

   Wiring the **datalog type** input of the File Dialog function to a dummy cluster of the same data type to be read causes the LabVIEW file dialog to display only directories and files of the appropriate data type. When the file is selected, it opens as a datalog file and a single data record is read.
7. Run the VI and test it with the `subset1.dat` file you created in Exercise 5-3.
8. Close the VI.
9. Open the State Machine with Login VI from Exercise 4-3.
10. Modify Case 3 as shown in the following block diagram.



   a. Delete the One Button Dialog from the case.
   b. Add the View Analysis File subVI so that it is called when you click the **View Data File** button.
   c. Save the State Machine with Login VI as `Completed Application.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

11. Display the front panel and run the Completed Application VI. Test all of the functionality of the application.
12. If time permits, complete the following challenge step, otherwise save and close the VI.

Challenge

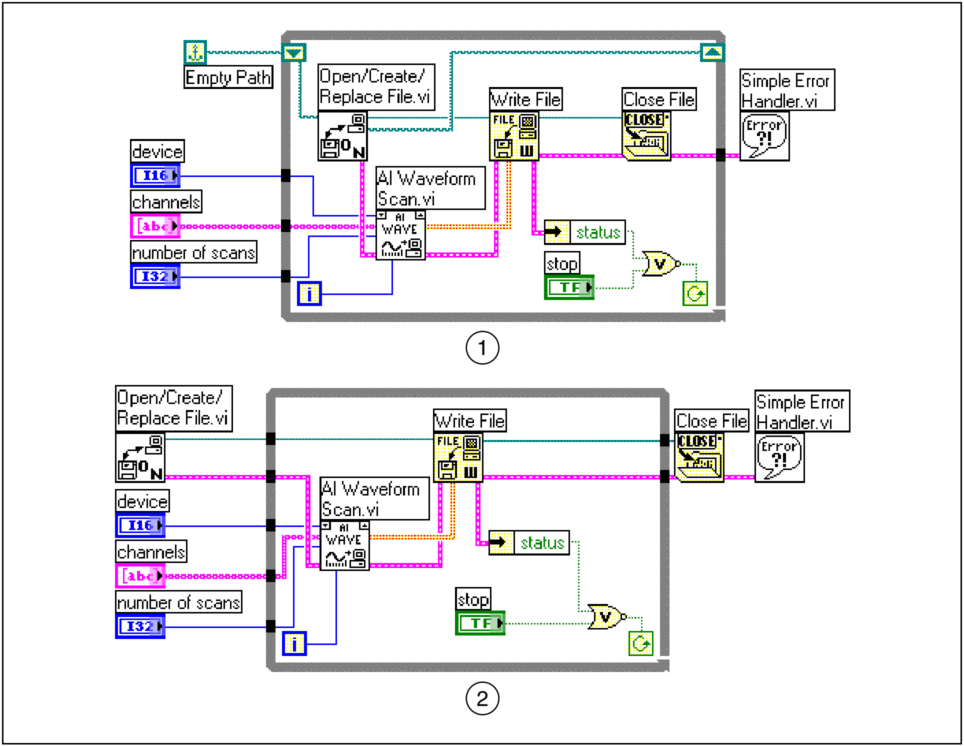
13. Use the Disable Controls VI you built in Exercise 3-5 to disable front panel controls on the Completed Application VI until the user logs in correctly.

End of Exercise 5-4

C. Disk Streaming

You also can use the low-level file I/O VIs and functions for disk streaming, which saves memory resources. Disk streaming is a technique for keeping files open while you perform multiple write operations, for example, within a loop. Although the high-level write operations are easy to use, they add the overhead of opening and closing the file each time they execute. Your VIs can be more efficient if you avoid opening and closing the same files frequently. Refer to Exercise 5-3 for an example of disk streaming.

The following examples show the advantages of using disk streaming. In the first example, the VI must open and close the file during each iteration of the loop. The second example uses disk streaming to reduce the number of times the VI must interact with the operating system to open and close the file. By opening the file once before the loop begins and closing it after the loop completes, you save two file operations on each iteration of the loop.



1 File I/O Example Without Using Disk Streaming	2 File I/O Example Using Disk Streaming

Avoid placing the high-level VIs located on the top row of the **Functions» All Functions»File I/O** palette in loops because these VIs perform open and close operations each time they run.

Summary, Tips, and Tricks

You can use the LabVIEW File I/O VIs and functions to work with text, binary, or datalog files. The same basic operations of Open File, Read File, Write File, and Close File work with all types of files.

Text Files

Text files store data as readable text characters. Text files are useful because almost all software applications and operating systems can read them. However, text files can be larger than necessary and therefore slower to access. It is also very difficult to perform random access file I/O with text files. You typically use text files when:

- Other users or applications need access to the data file.
- You do not need random access reading or writing in the data file.
- Disk space and file I/O speed are not crucial.

Binary Files

Binary data files store data in binary format without any conversion to text representation. Binary data files are generally smaller and thus faster to access. Random access file I/O presents no major difficulties. However, there is no industry-standard format for binary files. Thus, you must keep precise records of the exact data types and header information used in binary files. Use binary data files when:

- Other users or applications are unlikely to need access to your data.
- You need to perform random access file I/O in the data file.
- Disk space and file I/O speed are crucial.

Datalog Files

Datalog files are a special type of binary file for saving and retrieving complex data structures in LabVIEW. Like binary files, they have no industry-standard format. Use datalog files when:

- Your data is made up of mixed or complicated data types.
- Other users or applications are unlikely to need access to your data.
- Users who write VIs to access the data know the datalog structure.

Disk Streaming

Disk streaming is a technique of writing data to a file multiple times without closing the file after each write operation. Recall that the high-level file VIs open and close files each time they run, incurring unnecessary overhead in each iteration of the loop.

Additional Exercises

5-5 Write a VI that uses the Advanced File I/O functions to create a new file. Then, write to that file a single string composed of a string input by the user concatenated with a number converted to a text string using the Format Into String function. Select **File»Save As** to save the VI as `File Writer.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

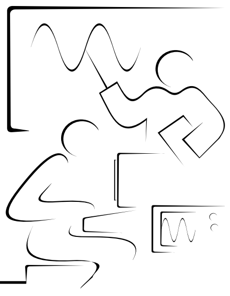
5-6 Write a VI that uses the Advanced File I/O functions to read the file created in Exercise 5-5. Select **File»Save As** to save the VI as `File Reader.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

Tip Use the EOF function, located on the **File I/O»Advanced File Functions** palette, to obtain the length of the written file.

Notes

Notes

Lesson 6
Project Management



This lesson describes some of the issues involved when building larger LabVIEW projects, including the design process, the organization of subVI components, and the process of creating a complete application.

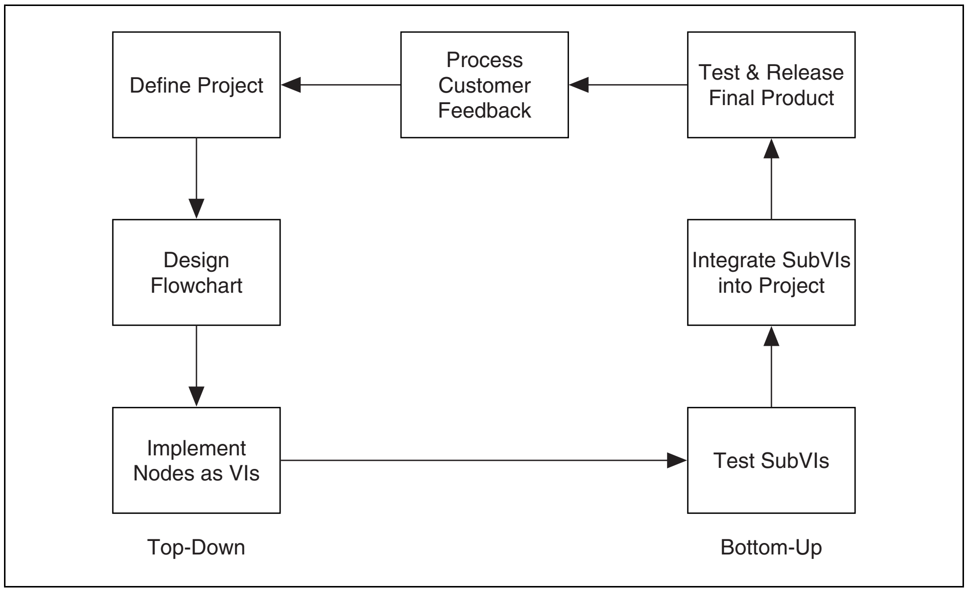
You Will Learn:

A. How to create a stand-alone executable for a LabVIEW application

B. LabVIEW features for managing project development

A. Creating an Executable (.exe) LabVIEW Application

The following illustration outlines a top-down design of an application, followed by a bottom-up implementation of the project as a series of subVIs.


Define Project
Process Customer Feedback
Test & Release Final Product
Design Flowchart
Integrate SubVIs into Project
Implement Nodes as VIs
Test SubVIs
Top-Down
Bottom-Up

In Lessons 1 through 5, you concentrated on the bottom elements of the development process, implementing and testing the subVIs that correspond to nodes of the project flowchart. You also have seen how to assemble these components into a larger application.

In this lesson, you will create an executable from the large application you have developed throughout the course. Your application is a basic data acquisition VI that meets the following criteria:

- Provides a menu-like user interface.
- Requires the user to log in with a correct name and password.
- Disables certain features if the user is not correctly logged in.
- Allows the user to configure the acquisition settings, including sample rate, number of samples, or to simulate data.
- Acquires waveform data with the specified user configuration.
- Allows the user to select a subset of the acquired data, analyze it, and save the analysis results to a file.
- Allows the user to load and view analysis results saved to disk.
- Stops the application with the click of a **Stop** button.

Note You can create stand-alone applications or shared libraries only in the LabVIEW Professional Development Systems or using the Application Builder add-on package.

LabVIEW Application Builder

Select **Tools»Build Application or Shared Library (DLL)** to use the Application Builder to create stand-alone applications and installers or shared libraries (DLLs) for VIs. The executable VI or shared libraries can include a hierarchy of VIs that you have created, or the VI can be configured to open and run any VI available to the user.

Required System Configuration

Applications or shared libraries that you create with the Application Builder generally have the same system requirements as the LabVIEW development system. Memory requirements vary depending on the size of the application created.

Creating a Stand-Alone Executable for an Application

Use the following tabs in the **Build Application or Shared Library (DLL)** dialog box to configure various settings for the application or shared library you want to build. After you define these settings, save them in a script so you can easily rebuild the application if necessary.

- From the **Target** tab, you can specify if you want to create a stand-alone executable or a shared library, the name of your application and the directory in which to create it. Optionally, you can choose to write subVIs to an external file if you want to keep the main application small.
- From the **Source Files** tab, you can define the VIs that make up your application. When you click **Add Top Level VI**, you add the main VI(s) for your application. You need to select only the top-level VI, and LabVIEW automatically includes all subVIs and related files, such as menu files or DLLs. If your VI dynamically calls any subVIs using the VI Server, LabVIEW cannot detect them automatically, so you must add them by clicking the **Add Dynamic VI** button. If you want to include any data files with your application, click the **Add Support File** button, and the data files automatically copy over to your application directory.
- From the **VI Settings** tab, specify modifications to make to your VIs as part of the build. You can choose to disable certain VI Properties. These settings only apply to the build process and do not affect your original source VIs. LabVIEW automatically creates your application as small as possible by removing debugging code, block diagrams, and unnecessary front panels. If you open a front panel dynamically using the VI Server, you must specify that the front panel is needed using the **VI Settings** tab.
- From the **Application Settings** tab, you can customize the features in your application. You can choose to specify the memory size for the Macintosh, or customize icons and ActiveX server features in Windows.
- **(Windows)** From the **Installer Settings** tab, you create an installer. The installer is written to the directory that contains your application.

Systems built using the Application Builder must include the LabVIEW Run-Time Engine.

Creating LabVIEW Applications

To create a professional, stand-alone application with VIs, you must understand four areas:

- The architecture of your application
- Programming issues particular to the application
- How to build your application
- How to build an installer for your application

Application Architecture

The application you have built throughout this course is a single top-level VI that runs when you launch the application and calls front panels from several subVIs. This is the most common and easiest architecture for building a stand-alone application.

Programming Issues

You should consider several programming issues when you are building VIs that end up as built applications. The first issue is to know what outside code is used for the application. For example, are you calling any system or custom DLLs or shared libraries? Are you going to process command line arguments? These are advanced examples that are beyond the scope of this course, but you need to consider them for the application.

A second issue is with the path names used in the VI. One example is when you use the VI Server capability to dynamically load and run VIs, which is described in the *LabVIEW Advanced: Performance & Communication* course. Once an application is built, the VIs are embedded in the executable. Suppose you have a VI named `test.vi` inside an application named `test.exe`. A Current VI's path primitive in `test.vi` returns `test.exe\test.vi` prepended with the full path of `test.exe`. Being aware of these issues will help you to build more robust applications in the future.

A last issue that affects the application you have currently built is that the top-level VI does not quit LabVIEW or close the front panel when it is finished executing. To completely quit and close the top-level VI, you must call the Quit LabVIEW function, located on the **Functions»All Functions» Application Control** palette, on the block diagram of the top-level VI.

Building the Application

You use the Application Builder in LabVIEW to create either an executable or a shared library (DLL) for your application. This course describes how to use the Application Builder in LabVIEW to build an executable. Refer to the *LabVIEW Help* or the *LabVIEW Advanced: Performance & Communication* course for information on how to build and use a shared library (DLL).

The LabVIEW Application Builder can package your application in one of two forms—as a single executable or as a single executable and one VI library. Depending upon how you want your application to appear to the end-user, as well as how complex the installation process can be, you might prefer one format to the other.

The default packaging for applications is a single executable file. All VIs that are part of the application are embedded in the executable file. These include top-level VIs, dynamic VIs, and all their subVIs. While this packaging is simple because it results in a single file, these files can become quite large depending on the number of VIs in your application.

The second option is to separate the application into an executable file and one VI library. In this packaging, the Application Builder embeds all top-level and dynamic VIs in the resulting executable file and all subVIs of these VIs are placed in a single VI library. While this package involves two files, the file that the end-user launches can be quite small.

Depending upon the nature of your application, it may require the presence of non-VI files to function correctly. Files commonly needed include a preferences (`.ini`) file for the application, the LabVIEW `serpdrv` file, and any help files that your VIs call. **(Windows and UNIX)** The LabVIEW `serpdrv` file is required for any application that uses serial port I/O. Run-time menu files and shared library files called using the Call Library Node function are not support files. The Application Builder includes run-time menu files in the main files for the application. It automatically stores any shared libraries needed in the support file directory for the application. External subroutines for CINs also are stored in the main files for the application.

Note Refer to the *LabVIEW Help* and the *LabVIEW Application Builder User Guide* for more detailed descriptions of how to use the Application Builder and make a preferences file for your application.

Building the Installer

The last phase in creating a professional, stand-alone application with your VIs is to create an installer. The LabVIEW Application Builder includes functionality for creating installers in Windows. Common tools for creating installers on a Macintosh are DragInstall and Vise. On UNIX systems, you can create a shell script to install your application. The installers you create with the LabVIEW Application Builder install all files that are part of the source files list. You must add all files that you want to install to this list. By specifying custom destinations for source files, you can create arbitrarily complex directory structures within the installation directory.

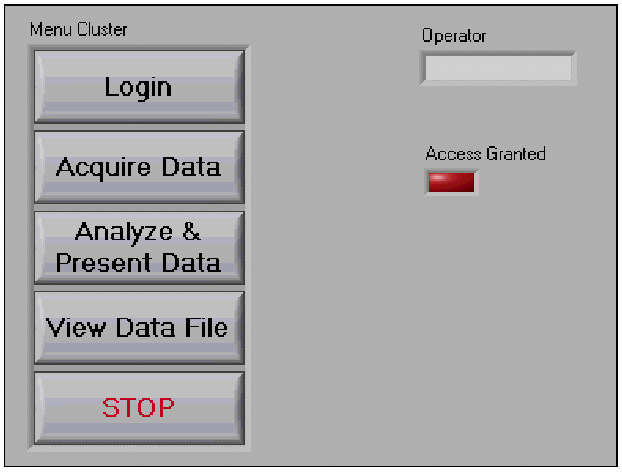
Exercise 6-1 My Application Executable

Objective: To create a stand-alone application with the Application Builder.

Note You must have the Application Builder properly installed to run this example. To determine whether it is installed, select the **Tools** menu. If the option **Build Application or Shared Library (DLL)** appears in the **Tools** menu, then the Application Builder is properly installed.

Front Panel

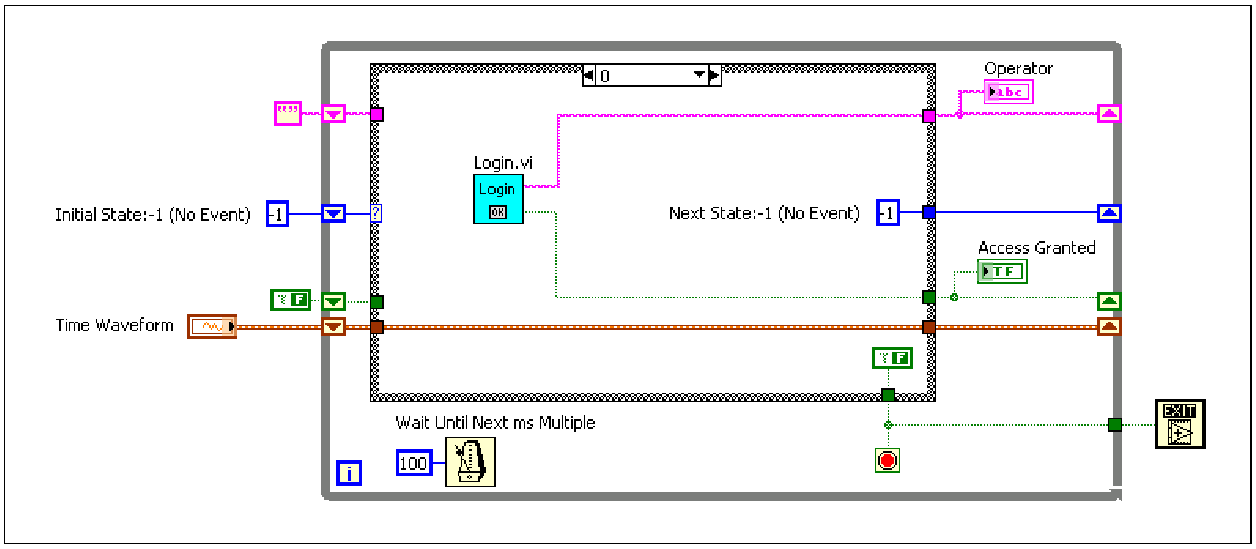
1. Open the Completed Application VI you created in Exercise 5-4. Modify the front panel to remove any visible comments.



2. Select **File»VI Properties** to display the **VI Properties** dialog box. Select **Window Appearance** from the top pull-down menu, then select **Top-level application window**. This gives the front panel a professional appearance when it is opened as an executable.
3. Select **File»Save As** to save the VI as `Application Exercise.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

Block Diagram

4. Open and modify the block diagram.



a. Place the Quit LabVIEW function, located on the **Functions»All Functions»Application Control** palette, on the block diagram. This function quits LabVIEW and quits the application after it has been built.

5. Save the VI under the same name.

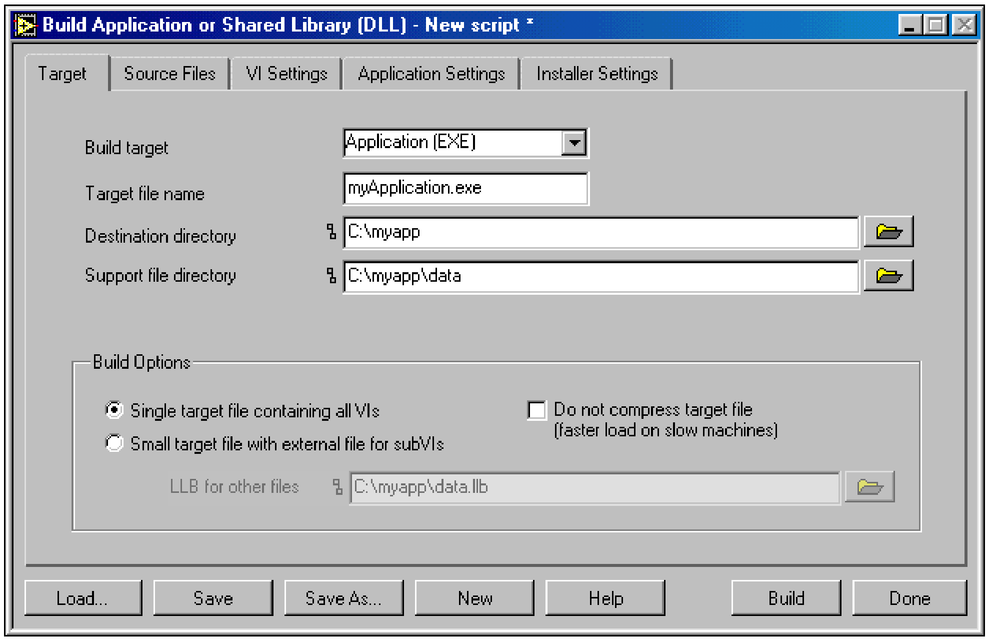
6. Open the front panel and run the VI. When you click the **Stop** button, the VI stops and you exit LabVIEW.

7. Restart LabVIEW and select **Tools»Build Application or Shared Library (DLL)** from the **LabVIEW** dialog box to open the **Build Application or Shared Library (DLL)** dialog box and begin building an executable from `Application Exercise.vi`.

Note Do not open the Application Exercise VI because the Application Builder cannot create an executable for a VI that is loaded into memory.

a. In the **Target** tab, select **Application(EXE)** from the **Build Target** pull-down menu, type `myApplication.exe` in the **Target file name** text box, and navigate to the `C:\myapp` directory for the **Destination directory** as shown in the following dialog box.

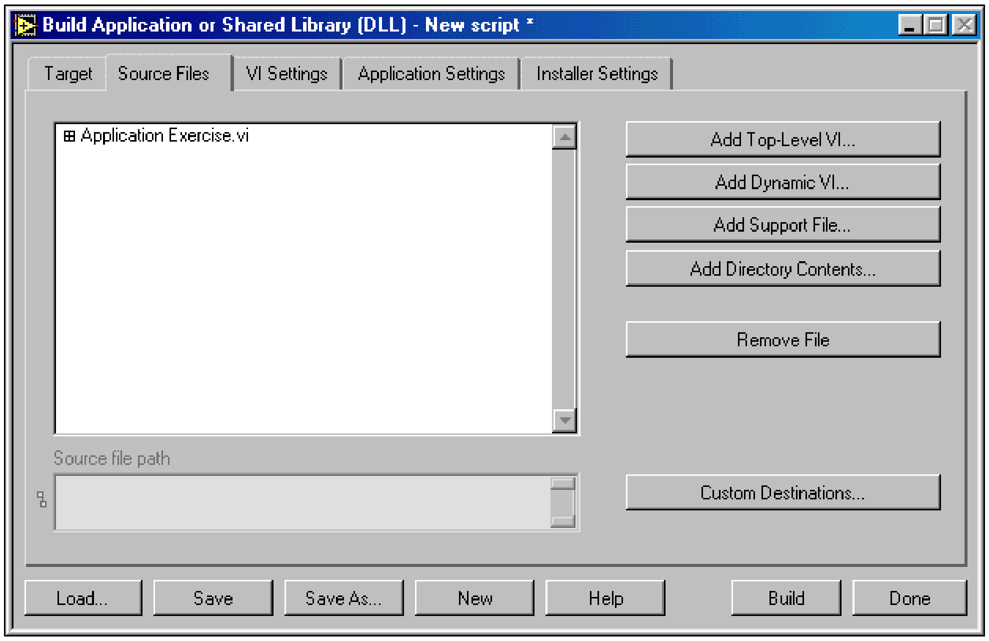
Tip Create the `myapp` directory by navigating to the `C:\` directory and clicking the **Create New Folder** icon in the **Select a destination directory** dialog box.


Build Application or Shared Library (DLL) - New script *

Build target: Application (EXE)
Target file name: myApplication.exe
Destination directory: C:\myapp
Support file directory: C:\myapp\data

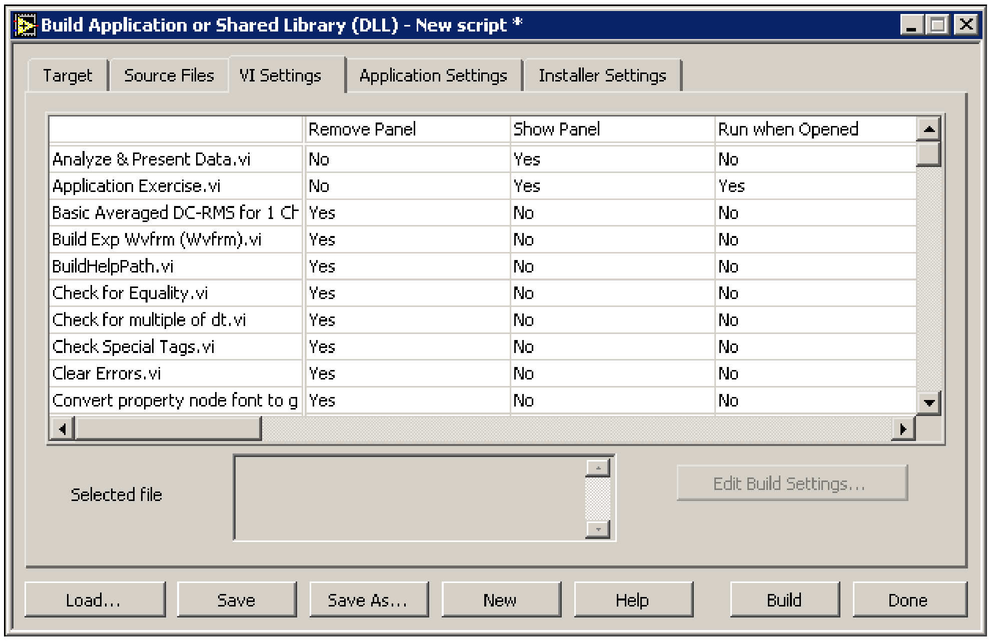
Build Options
[✓ Single target file containing all VIs]
Small target file with external file for subVIs
Do not compress target file (faster load on slow machines)
LLB for other files: C:\myapp\data.llb

b. Click the **Source Files** tab and click the **Add Top-Level VI** button. Add `Application Exercise.vi` as shown in the following dialog box.


Build Application or Shared Library (DLL) - New script *

⊞ Application Exercise.vi

c. Click the **VI Settings** tab. Leave these settings at their default values—the top-level VI runs when opened and the block diagrams and front panels are only saved if they are necessary. Examine the settings to ensure they are similar to the following dialog box.

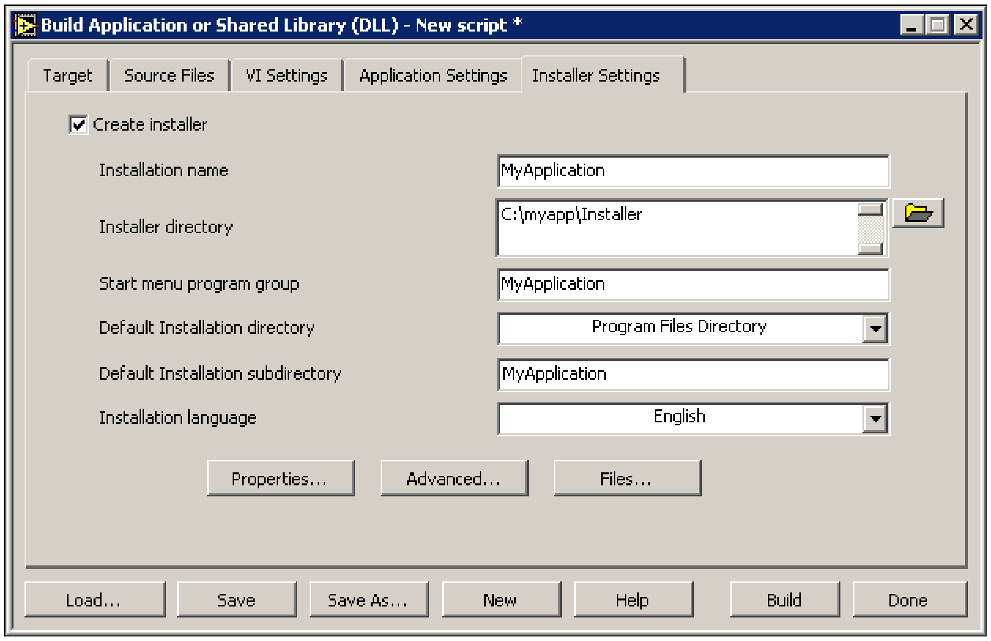


	Remove Panel	Show Panel	Run when Opened
Analyze & Present Data.vi	No	Yes	No
Application Exercise.vi	No	Yes	Yes
Basic Averaged DC-RMS for 1 Ch	Yes	No	No
Build Exp Wvfrm (Wvfrm).vi	Yes	No	No
BuildHelpPath.vi	Yes	No	No
Check for Equality.vi	Yes	No	No
Check for multiple of dt.vi	Yes	No	No
Check Special Tags.vi	Yes	No	No
Clear Errors.vi	Yes	No	No
Convert property node font to g	Yes	No	No

Selected file

d. Click the **Application Settings** tab. This is where you would enable ActiveX settings or give your application a custom icon. Leave the icon as the default LabVIEW icon. Do not change any of these settings.

e. Click the **Installer Settings** tab. Build a distribution kit for your application that installs into the `C:\Program Files\MyApplication` directory. Configure the **Installer Settings** tab as shown in the following dialog box.



☑ Create installer

Installation name	MyApplication
Installer directory	C:\myapp\Installer
Start menu program group	MyApplication
Default Installation directory	Program Files Directory
Default Installation subdirectory	MyApplication
Installation language	English

f. Click the **Build** button. The files associated with the installer are compressed and stored in the `C:\myapp\installer` directory. A `setup.exe` file is created as well, which can be used to install the files. All of these files could be copied to CDs to transfer the application to another system. The LabVIEW Run-Time DLL installer is included by default. The executable for your application is also built and is called `myApplication.exe`, as defined on the **Target** tab.

g. Click the **Done** button in the **Build Application or Shared Library (DLL)** dialog box to close the utility. When prompted to save a script so you can build this application again, select **Yes** and name the script `myapp.bld`. If you make changes to the original application and want to rebuild an executable and installer with the same settings, you can open this script file using the **Load** button in the **Build Application or Shared Library (DLL)** dialog box.

8. Run `myApplication.exe` from the `C:\myapp` directory. Application Exercise should open its front panel and run automatically. Operate the VI to make sure all the settings you chose are working. Close the application when you are finished.

9. Run the `setup.exe` file in the `C:\myapp\Installer` directory. You should be guided through a setup process. The executable is created inside the `C:\Program Files\MyApplication` directory.

10. To run the application, select **Start»Programs»MyApplication» MyApplication**.

End of Exercise 6-1

B. LabVIEW Features for Project Development

LabVIEW provides several features you can use to manage your projects more efficiently.

LabVIEW VI Libraries (LLBs)

You can save VIs as individual files, or you can group several VIs together and save them in a VI library. VI library files end with the extension `.llb`. You usually want to save VIs as individual files, organized in directories, especially if multiple developers are working on the same project.

The following list describes reasons to save VIs as individual files:

- You can use the file system to manage the individual files.
- You can use subdirectories.
- You can store VIs and controls in individual files more robustly than you can store your entire project in the same file.
- You can use the Professional Development System built-in source code control tools or third-party source code control tools.

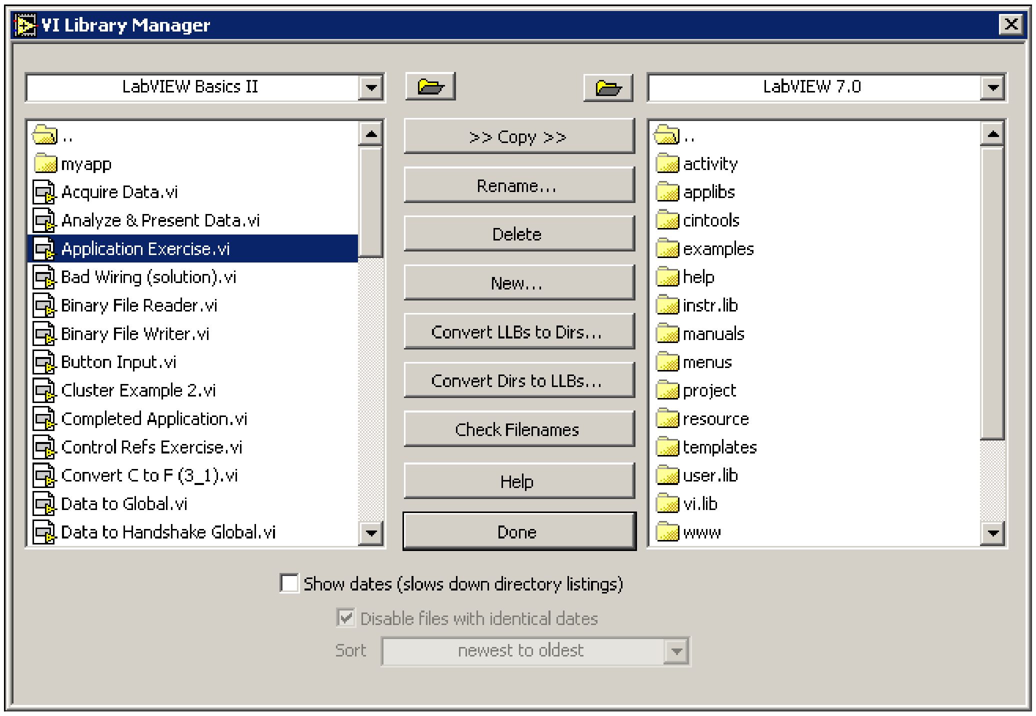
The following list describes reasons to save VIs as libraries:

- You can use up to 255 characters to name your files.

  (Mac OS) Mac OS 9.*x* or earlier limits you to 31 characters for filenames.
- You can transfer a VI library to other platforms more easily than you can transfer multiple individual VIs.
- You can slightly reduce the file size of your project because VI libraries are compressed to reduce disk space requirements.
- You can mark VIs in a library as top-level so when you open the library, LabVIEW automatically opens all the top-level VIs in that library.

Saving changes to a VI in a library takes longer than saving changes to an individual VI, because the operating system must write the changes to a larger file. Saving changes to a large library also can increase memory requirements and decrease performance. Try to limit the size of each library to approximately 1 MB.

The **VI Library Manager** utility enables you to copy, rename, and delete VIs, whether they are located in LLBs or not. The VI Library Manager also can convert existing LLBs into files in a subdirectory.



VI History

One of the most useful LabVIEW tools for team-oriented development is the **History** window. Use the **History** window in each VI to display the development history of the VI, including revision numbers. The revision number starts at zero and increases incrementally every time you save the VI. Record and track the changes you make to the VI in the **History** window as you make them. Select **Tools»VI Revision History** to display the **History** window. You also can print the revision history.

The **Revision History** page of the **VI Properties** dialog box and the **Revision History** page of the **Tools»Options** dialog box contain similar options. Use the **VI Properties** dialog box to set options for the current VI. Use the **Options** dialog box to set options for all new VIs.

VI Hierarchy

One of the most important advantages of separating your main application into subVIs is that you save memory. In addition, the responsiveness of the LabVIEW editor improves because smaller VIs are easier to handle. Using subVIs makes the high-level block diagram easy to read, debug, understand, and maintain.

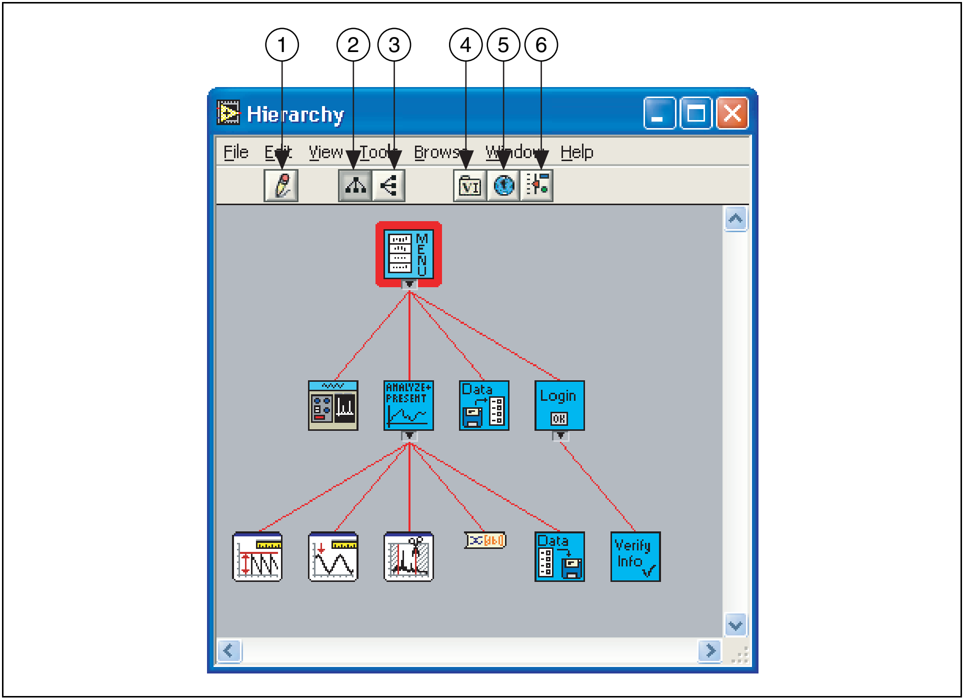
Therefore, try to keep the block diagram for your top-level VI under 500 KB in size. In general, your subVIs should be significantly smaller. To check the size of a VI, select **File»VI Properties** and select **Memory Usage** from the top pull-down menu. Typically, you should break a VI into several subVIs if the block diagram for your VI is too large to fit entirely on the screen.

If you find that the block diagram for a VI is getting too large, convert a section of a VI into a subVI by using the Positioning tool to select the section of the block diagram you want to reuse and selecting **Edit»Create SubVI**.

The **Hierarchy** window displays a graphical representation of the calling hierarchy for all VIs in memory, including type definitions and global variables. Select **Browse»Show VI Hierarchy** to display the **Hierarchy** window. Use this window to view the subVIs and other nodes that make up the current VI.

Use the toolbar at the top of the **Hierarchy** window to show or hide various categories of objects used in the hierarchy, such as global variables or VIs shipped with LabVIEW, as well as whether the hierarchy expands horizontally or vertically. A VI that contains subVIs has an arrow button on its border. Click this arrow button to show or hide subVIs. A red arrow button appears when all subVIs are hidden. A black arrow button appears when all subVIs are displayed.

The following **Hierarchy** window contains the hierarchy of the application you completed in the previous exercise. The VIs from the LabVIEW `vi.lib` directory are not shown. Right-click a blank area of the window and select **Show All VIs** from the shortcut menu to show the entire hierarchy.



1 Redo Layout	4 Include VI Lib
2 Vertical Layout	5 Include Globals
3 Horizontal Layout	6 Include Type Definitions

As you move the Operating tool over objects in the **Hierarchy** window, LabVIEW displays the name of each VI. Double-click a VI or subVI node in the **Hierarchy** window to display the front panel of that VI. You also can locate a VI in the hierarchy by typing the name of the node you want to find anywhere in the window. As you type the text, the search string appears, displaying the text as you type. LabVIEW highlights the node with a name that matches the search string. You also can find a node in the hierarchy by selecting **Edit»Find**.

Use the **Hierarchy** window as a development tool when planning or implementing your project. For example, after developing a flowchart of the VIs required for an application, you can create, from the bottom of the hierarchy up, each of these VIs so that they have all necessary inputs and outputs on their front panel, and the subVIs they call on their block diagrams. This builds the basic application hierarchy, which now appears in the **Hierarchy** window. You then can begin to develop each subVI, perhaps color-coding their icons, which will also be colored in the **Hierarchy** window, to reflect their status. For example, white icons could represent untouched VIs, red icons could represent subVIs in development, and blue icons could represent completed VIs.

Using Online Help in Your LabVIEW Applications

As you put the finishing touches on your application, you should provide online help to the user. Create descriptions for VIs and their objects, such as controls and indicators, to describe the purpose of the VI or object and to give users instructions for using the VI or object.

Use the following functions, located on the **Functions»All Functions» Application Control»Help** palette, to programmatically show or hide the **Context Help** window and link from VIs to HTML files or compiled help files:

- Use the Get Help Window Status function to return the status and position of the **Context Help** window.
- Use the Control Help Window function to show, hide, or reposition the **Context Help** window.
- Use the Control Online Help function to display the table of contents, jump to a specific point in the file, or close the online help.Use the Open URL in Default Browser VI to display a URL or HTML file in the default Web browser.

Comparing VIs

The LabVIEW Professional Development System includes a utility to determine the differences between two VIs loaded into the memory. From the LabVIEW pull-down menu, select **Tools»Compare»Compare VIs** to display the **Compare VIs** dialog box.

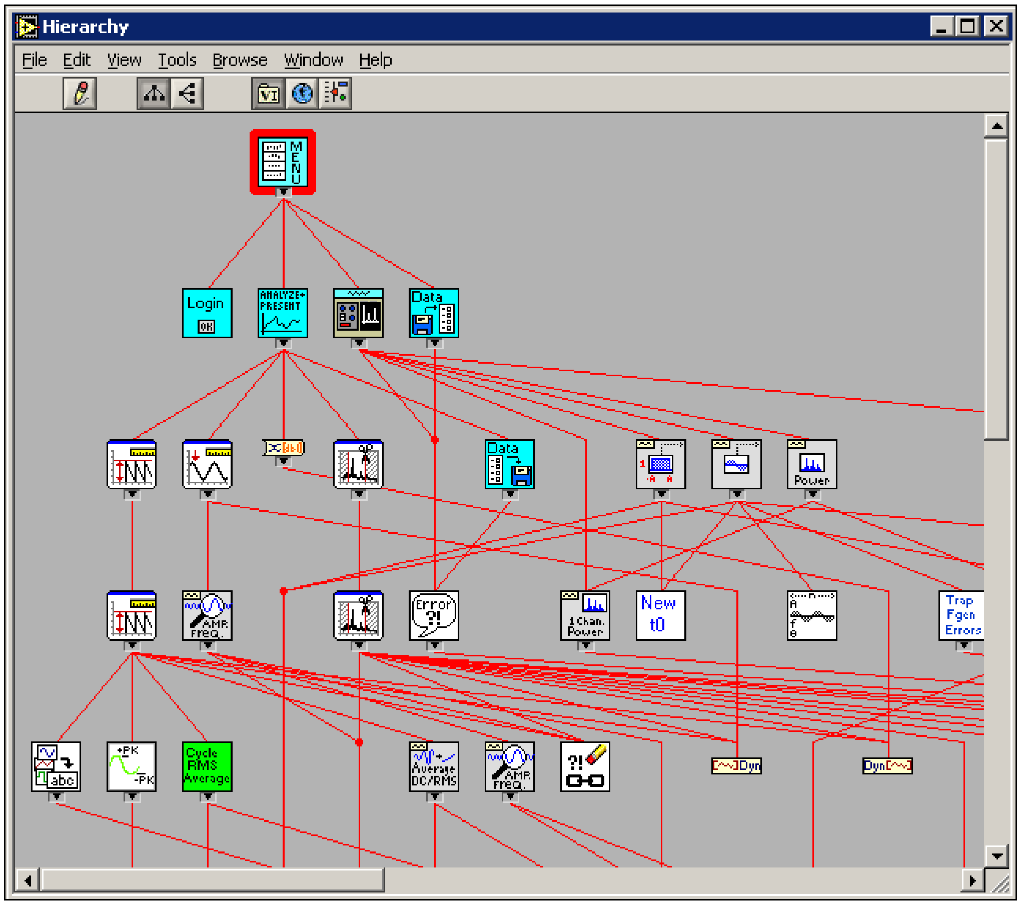
From this dialog box, you can select the VIs you want to compare, as well as the characteristics of the VIs to check. When you compare the VIs, both VIs display, along with a **Differences** window that lists all differences between the two VIs. In this window, you can select various differences and details to view, which can be circled for clarity.

Exercise 6-2 LabVIEW Project Management Tools

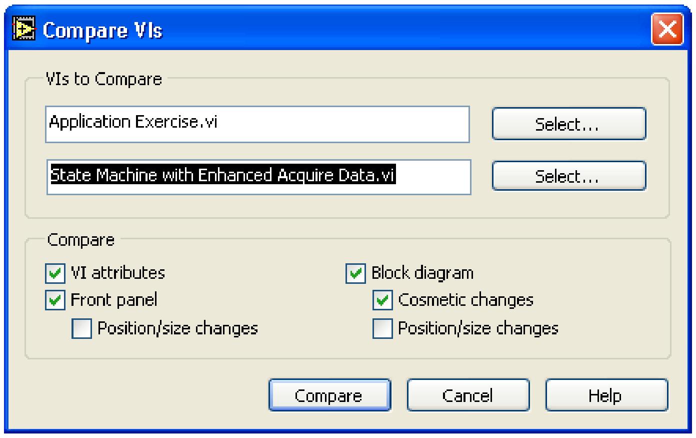
Objective: To examine some of the built-in LabVIEW features for handling applications.

In this exercise, you explore some of the features built into LabVIEW for handling applications.

1. Open the Application Exercise VI you created in Exercise 6-1. Close any other VIs loaded into memory.
2. Select **Tools»VI Revision History** to open the **History** window for the VI.
3. Click the **Reset** button to clear the current history. Click **Yes** to confirm the deletion of the history and reset the revision number.
4. In the **Comment** text box of the **History** window, type `Initial Application Created` and click the **Add** button. Your comment appears in the **History** text box, along with a date and time stamp. Close the **History** window.
5. Select **Browse»Show VI Hierarchy**. The applications hierarchy appears.



6. Experiment with expanding and collapsing the hierarchy. Notice that as you click the small black and red arrows in the hierarchy, they expand or collapse branches of the hierarchy. You might see some icons with a red arrow by them, indicating that they call one or more subVIs. In addition, you might also see icons with a blue arrow next to them, which occurs when a subVI is called from multiple places in an application, but not all calls are currently indicated in the hierarchy.
7. Examine the operation of the buttons in the hierarchy toolbar. Notice how you can arrange the hierarchy using the **Layout** buttons or by dragging the icons, or include various application components using the **Include** buttons. Use **Redo Layout** to redraw the window layout to minimize line crossing and maximize symmetry.
8. Double-click any subVI icon in the hierarchy to display the appropriate subVI. Close the subVI you selected, and close the **Hierarchy** window.
9. Open the State Machine with Enhanced Acquire Data VI you completed in Exercise 2-5, change to the front panel of the Application Exercise VI, and then select **Tools»Compare»Compare VIs** to display the **Compare VIs** dialog box.



VIs to Compare

Application Exercise.vi

State Machine with Enhanced Acquire Data.vi

Compare

[✓] VI attributes
[✓] Front panel
[ ] Position/size changes
[✓] Block diagram
[✓] Cosmetic changes
[ ] Position/size changes

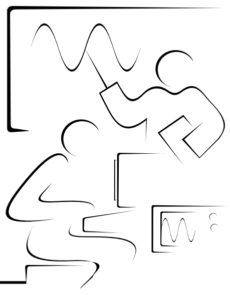
10. Using the **Select** button, make sure that the correct VIs are listed in the **VIs to Compare** box, and that the **Compare** options are set as shown in the previous dialog box.
11. Click **Compare** to display the **Differences** window and tile the two VIs. Place a checkmark in the **Circle Differences** checkbox in the **Differences** window. Then, select a difference from the Differences listbox, select a detail from the Details listbox, and then click **Show Detail**. The difference between the two VIs is highlighted. Examine the various differences between the two VIs and then close the **Differences** window.
12. Close both VIs. Do not save any changes.

End of Exercise 6-2

Summary, Tips, and Tricks

- LabVIEW features the Application Builder, which enables you to create stand-alone executables or shared libraries (DLLs). The Application Builder is available in the Professional Development Systems, or as an add-on package.
- Creating a professional, stand-alone application with your VIs involves four areas of understanding:
  - The architecture of your application
  - The programming issues particular to the application
  - The application building process
  - The installer building process
- LabVIEW has several features to assist you and your coworkers in developing your projects, such as the **VI Revision History** window to record comments and modifications to a VI, and the user login, which, when used with VI Revision History, records who made changes to a VI. You can access the **VI Revision History** window at any time by selecting **Tools»VI Revision History**.
- The **Hierarchy** window provides a quick, concise overview of the VIs used in your project.
- The comparison feature identifies the differences between two VIs.

Notes



Lesson 7
Remote Data Management

This lesson describes the capabilities built into LabVIEW that allow you to easily work with networks. You can pass data between LabVIEW VIs on interconnected computers, and control VIs running on one computer from another computer. LabVIEW provides the high-level tools needed to make these interconnections so you can acquire, analyze, and present data from anywhere in your business.

You Will Learn:

A. How to use DataSocket

B. How to view and control front panels in LabVIEW

C. How to view and control front panels from a Web browser

D. The LabVIEW Web Server

A. DataSocket

Use National Instruments DataSocket technology to share live data with other VIs and other applications, such as National Instruments Measurement Studio, on the Web, or on your local computer. DataSocket pulls together established communication protocols for measurement and automation in much the same way a Web browser pulls together different Internet technologies.

How Does DataSocket Work?

DataSocket consists of two components, the DataSocket API and the DataSocket Server. The DataSocket API presents a single user interface for communicating with multiple data types from multiple programming languages. The DataSocket Server simplifies Internet communication by managing TCP/IP programming for you.

DataSocket API

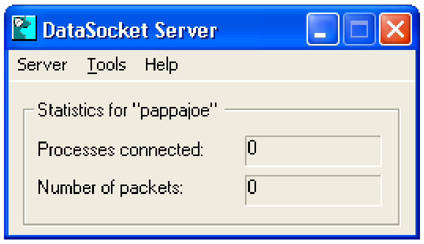
DataSocket is a single, unified, end-user API based on URLs for connecting to measurement and automation data located anywhere, whether on a local computer or on the Internet. It is a protocol-independent, language-independent, and OS-independent API designed to simplify binary data publishing. The DataSocket API is implemented so you can use it in any programming environment and on any operating system.

The DataSocket API automatically converts your measurement data into a stream of bytes that is sent across the network. The subscribing DataSocket application automatically converts the stream of bytes back into its original form. This automatic conversion eliminates network complexity, which accounts for a substantial amount of code that you must write when using TCP/IP libraries.

DataSocket Server

The DataSocket Server is a lightweight, stand-alone component with which programs using the DataSocket API can broadcast live measurement data at high rates across the Internet to several remote clients concurrently.

The DataSocket Server simplifies network TCP programming by automatically managing connections to clients. Access the DataSocket Server by selecting **Start»Programs»National Instruments» DataSocket»DataSocket Server**. When you select the DataSocket Server, it opens the following window and begins running.



Statistics for "pappajoe"

Processes connected: 0

Number of packets: 0

As shown previously, the DataSocket Server keeps track of the number of clients connected to it as well as how many packets of data have been exchanged. You can select to have the DataSocket Server run hidden by selecting **Server»Hide DataSocket Server**.

Broadcasting data with the DataSocket Server requires three components—a publisher, the DataSocket Server, and a subscriber. A publishing application uses the DataSocket API to write data to the server. A subscribing application uses the DataSocket API to read data from the server. Both the publishing and the subscribing applications are clients of the DataSocket Server. The three components can reside on the same machine, but more often the three components run on different machines. The ability to run the DataSocket Server on another machine improves performance and provides security by isolating network connections from your measurement application. The DataSocket Server restricts access to data by administering security and permissions. With DataSocket, you can share confidential measurement data over the Internet while preventing access by unauthorized viewers.

A URL to Any Data Source

DataSocket technology provides access to several input and output mechanisms from the front panel through the DataSocket Connection dialog box. Right-click a front panel object and select **Data Operations» DataSocket Connection** from the shortcut menu to display the **DataSocket Connection** dialog box. You publish (write) or subscribe to (read) data by specifying a URL, in much the same way you specify URLs in a Web browser.

URLs use communication protocols, such as `dstp`, `ftp`, and `file`, to transfer data. The protocol you use in a URL depends on the type of data you want to publish and how you configure your network. The *how* encoded in the first part of the URL is called the access method of protocol. Web browsers typically use several access methods, such as HTTP, HTTPS (encrypted HTTP), FTP (file transfer protocol), and FILE (for reading files on your local machine). DataSocket takes the same approach for measurement data. For example, DataSocket can use the following URL to connect to a data item: `dstp://mytestmachine/wave1`. The `dstp` prefix tells DataSocket to open a data socket transfer protocol connection to the test machine and retrieve a signal named `wave1`. If `file` is the URL prefix, DataSocket retrieves the data from a file instead of the DataSocket server.

Direct DataSocket Connection to Any Panel Object

Use front panel DataSocket connections to publish or subscribe to live data in a front panel object. When you share the data of a front panel object with other users, you publish data. When users retrieve the published data and view it on their front panel, users subscribe to the data.

In the **DataSocket Connection** dialog box, enter a valid URL in the **Connect To** field, select whether you want to publish, subscribe, or both, place a checkmark in the **Enabled** checkbox, and click the **Attach** button. The front panel object is available at the specified URL in the DataSocket Server. If the connection to the DataSocket Server is successful, a small green indicator appears next to the front panel object on your VI. If the connection is not successful, a small red indicator appears. If LabVIEW cannot connect to the server, a small gray indicator appears.

DataSocket Functions

From the block diagram, you can programmatically read or write data using the DataSocket functions located on the **Functions»All Functions» Communication»DataSocket** palette.

You can do all the basic DataSocket operations with the following VIs and functions:

- The DataSocket Open function opens a DataSocket connection specified in the URL with a particular mode. The mode can be either Read, Write, ReadWrite, BufferedRead, or BufferedReadWrite. This course describes only the Read and Write modes.
- The DataSocket Write function writes data to a specified URL. The data can be in any format or LabVIEW data type.
- The DataSocket Read function reads data from a specified URL connection and returns the data, which you can publish to another front panel object or pass to another function.
- The DataSocket Close function closes a DataSocket connection specified by the DataSocket Connection Refnum.
- Use the DataSocket Select URL VI only when you do not know the URL for an object and you want to search for a data source or target from a dialog box.

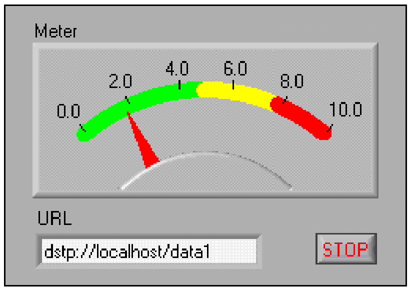
Exercise 7-1 DS Generate Data VI and DS Read Data VI

Objective: To build two VIs that use DataSocket to transfer data.

Build a VI that generates a random number and displays this value in a meter and sends the data to a DataSocket URL. Build a second VI that reads the DataSocket URL and displays the value in a slide. Use the automatic publish and subscribe features to send the data between the VIs through a DataSocket connection.

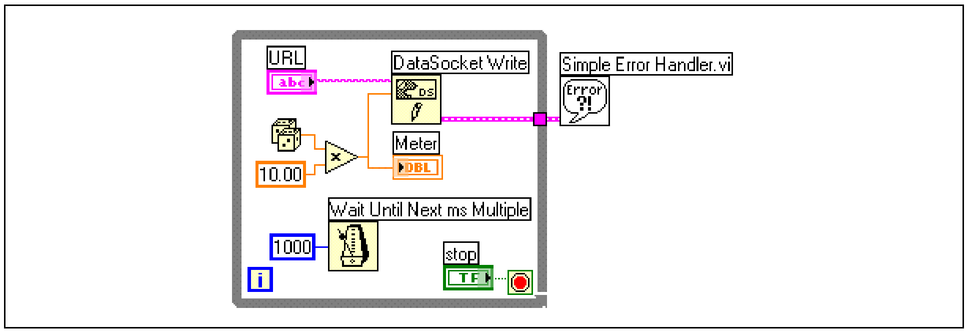
Front Panel

1. Open a new VI and build the following front panel. The URL object is a string control.



Block Diagram

2. Open and build the following block diagram.



a. Place a While Loop, located on the **Functions»Execution Control** palette, on the block diagram.

b. Place the DataSocket Open function, located on the **Functions»All Functions»Communications»DataSocket** palette, on the block diagram.This function opens a DataSocket connection. Right-click the **mode** input and select **Create»Constant** from the shortcut menu to create the enumerated constant. Select **Write** for the mode with the Operating tool.

c. Place the Wait Until Next ms Multiple function, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram. This function causes the While Loop to execute once per second. Right-click the **input** terminal and select **Create»Constant** from the shortcut menu. Enter a value of `1000` in the constant.

d. Place the Random Number (0-1) function, located on the **Functions»Arithmetic & Comparison»Express Numeric** palette, on the block diagram.This function creates a random number between zero and one.

e. Place the Multiply function, located on the **Functions»Arithmetic & Comparison»Express Numeric** palette, on the block diagram.This function multiplies two numbers together and is used in this exercise to scale the random number to be between zero and 10. Right-click the bottom input and select **Create»Constant** from the shortcut menu. Enter a value of `10` in the constant.

f. Place the DataSocket Write function, located on the **Functions»All Functions»Communication»DataSocket** palette, on the block diagram.This function writes the random data value to the specified URL.

g. Place the DataSocket Close function, located on the **Functions»All Functions»Communication»DataSocket** palette, on the block diagram.This function closes a DataSocket connection specified by the DataSocket Connection Refnum.

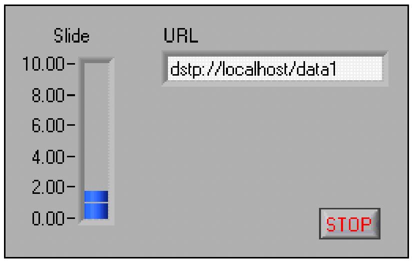
h. Place the Simple Error Handler VI, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram.This VI opens a dialog box if an error occurs and displays the error information.

3. Save this VI as `DS Generate Data.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

Build a second VI to read the random value.

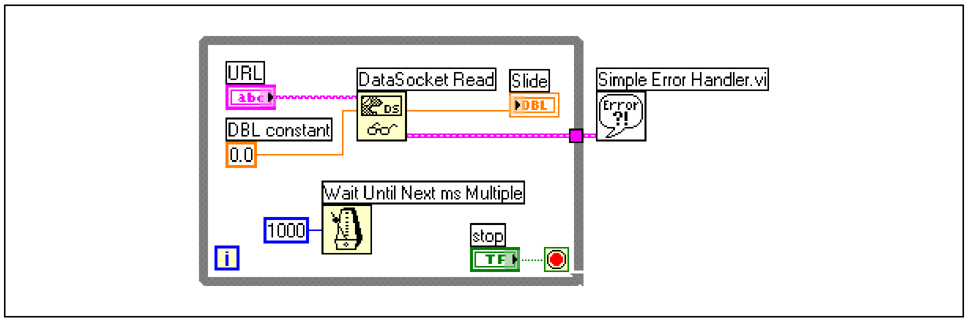
Front Panel

4. Open a new VI and build the following front panel. The URL object is a string control. To display the scale on the slide indicator, right-click the slide and select **Scale»Style**.



Block Diagram

5. Open and build the following block diagram.



a. Place a While Loop, located on the **Functions»Execution Control** palette, on the block diagram.

b. Place the DataSocket Open function, located on the **Functions»All Functions»Communications»DataSocket** palette, on the block diagram. This function opens a DataSocket connection. Right-click the **mode** input and select **Create»Constant** to create the enumerated constant. Click the constant with the Operating tool and set the mode to **Read**.

c. Place the Wait Until Next ms Multiple function, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram.This function causes the While Loop to execute once per second. Create the constant by right-clicking the **input** terminal and selecting **Create»Constant** from the shortcut menu.

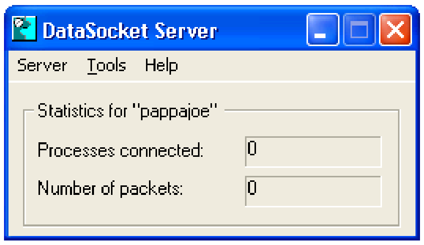
d. Place the Numeric constant, located on the **Functions»Arithmetic & Comparison»Express Numeric** palette, on the block diagram. This constant creates the correct data type to read the value through DataSocket. Right-click the constant and select **Representation» DBL** from the shortcut menu.

e. Place the DataSocket Read function, located on the **Functions»All Functions»Communication»DataSocket** palette, on the block diagram.This function reads the random data value from the specified URL.

f. Place the DataSocket Close function, located on the **Functions»All Functions»Communication»DataSocket** palette, on the block diagram.This function closes a DataSocket connection specified by the DataSocket Connection Refnum.

g. Place the Simple Error Handler VI, located on the **Functions»All Functions»Time & Dialog** palette, on the block diagram.This VI opens a dialog box if an error occurs and displays the error information.

6. Save this VI as `DS Read Data.vi` in the `C:\Exercises\LabVIEW Basics 2` directory.
7. Position the front panels of the DS Generate Data VI and DS Read Data VI so that you can see both front panels.
8. Start the DataSocket Server by selecting **Start»Programs»National Instruments»DataSocket»DataSocket Server**. The **DataSocket Server** window appears, similar to the following example.



Statistics for "pappajoe"

Processes connected: 0

Number of packets: 0

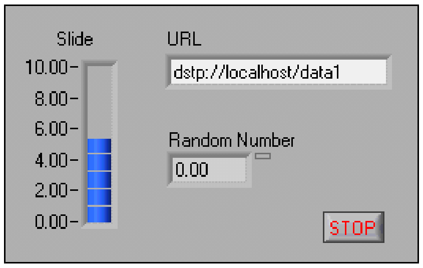
9. Return to the two VI front panels and make sure that the same URLs have been entered for both VIs.
   - **dstp**—The DataSocket transfer protocol.
   - **localhost**—The current computer you are using.
   - **data1**—The name given to the random number you will be sending.
10. Run the DS Generate Data VI and the DS Read Data VI.

    The **DataSocket Server** window shows one process connected, and the Number of packets value increments each second as the meter and slide show the same random numbers.
11. Stop both VIs when you are finished.

Modify the VIs to use the automatic publish and subscribe capabilities of front panel objects.

Front Panel

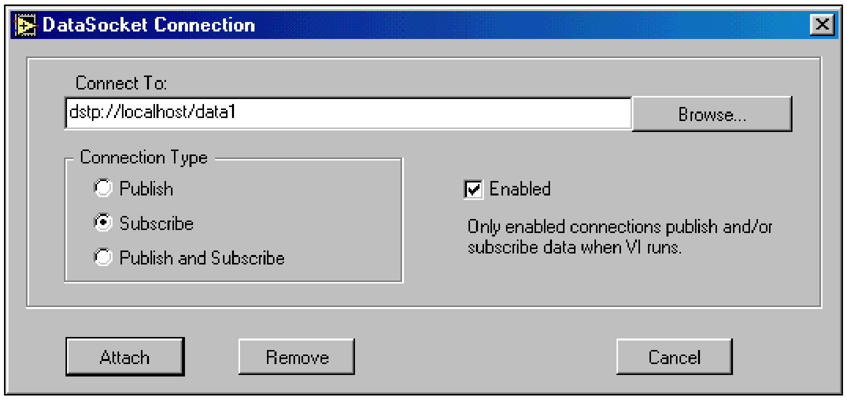
12. Place a numeric indicator on the front panel of the DS Read Data VI as shown in the following example.



dstp://localhost/data1

0.00

13. Right-click the numeric indicator and select **Data Operations» DataSocket Connection** from the shortcut menu. Enter the values shown in the following example.



Connect To: dstp://localhost/data1

Connection Type
Publish
[✓ Subscribe]
Publish and Subscribe

[✓ Enabled]

Only enabled connections publish and/or subscribe data when VI runs.

14. Click the **Attach** button. A small gray rectangle appears to the top right side of the numeric indicator to indicate that the DataSocket connection is not active.
15. Run the DS Generate Data and DS Read Data VIs again.

    The rectangle next to the Random Number indicator turns green, and the value matches the values shown in the meter and the slide.

Note If your computer is on a network with the other computers used in class, you can enter URLs for other machines on the network and transfer the values between classroom computers. Remember that you can use any programming language or operating system with DataSocket connections. Refer to the National Instruments DataSocket Web page at `ni.com/datasocket` for more information.

16. Stop and close both VIs and the DataSocket Server when you are finished.

End of Exercise 7-1

B. LabVIEW Web Server

Use the LabVIEW Web Server to create HTML documents and publish front panel images on the Web. You can control browser access to the published front panels and configure which VIs are visible on the Web.

Note Use the LabVIEW Enterprise Connectivity Toolset to control VIs on the Web and to add more security features to VIs you publish on the Web. Refer to the National Instruments Web site at `ni.com` for more information about this toolset.

You must enable the Web Server in the **Web Server: Configuration** page of the **Options** dialog box before you can publish VIs on the Web. You also can enable the Web Server with the Web Publishing Tool, described in the following section. The VIs must be in memory before you publish them. The default Web Server configuration is suitable for most applications. If you need to change this default configuration, refer to the *LabVIEW Help*.

Use the **Web Server: Browser Access** page of the **Options** dialog box to configure which browser addresses can view your VI front panels. Create a **Browser Access List** that allows and denies access to individual browser addresses. When a Web browser attempts to obtain a VI front panel image, the Web Server compares the browser address to the entries in the **Browser Access List** to determine whether it should grant access. If an entry in the **Browser Access List** matches the browser address, the Web Server permits or denies access based on how you set up the entry. By default, all browsers have access to the LabVIEW Web Server.

Use the **Web Server: Visible VIs** page of the **Options** dialog box to configure which VIs are visible on the Web. Create a **Visible VIs** list that allows and denies access to individual VIs, groups of VIs, or directory paths. When a Web browser attempts to obtain a VI front panel image, the Web Server compares the VI name to the entries in the **Visible VIs** list to determine whether it should grant access. If an entry in the **Visible VIs** list matches the VI name, the Web Server permits or denies access to that VI image based on how you set up the entry. By default, the front panel images of all VIs are visible.

Web Publishing Tool

Select **Tools»Web Publishing Tool** to use the Web Publishing Tool to create an HTML document and embed static or animated images of the front panel. You also can embed images of the front panel in an existing HTML document.

Click the **Instructions** button to display the **Instructions** window. This window contains information about how to add a title to your HTML file and how to add text before and after your VI front panel. Enter a VI name in the **VI Name** field or select **Browse** from the **VI Name** pull-down menu and navigate to a VI.

If you want to preview the document in your default browser, click the **Start Web Server** button and then click the **Preview in Browser** button. If the **Start Web Server** button is dimmed, the Web Server is already running. Click the **Save to Disk** button to save the title, text, and VI front panel image in an HTML document. If you want to view the document from a remote computer, save the HTML document in the Web Server root directory, usually `labview\www`.

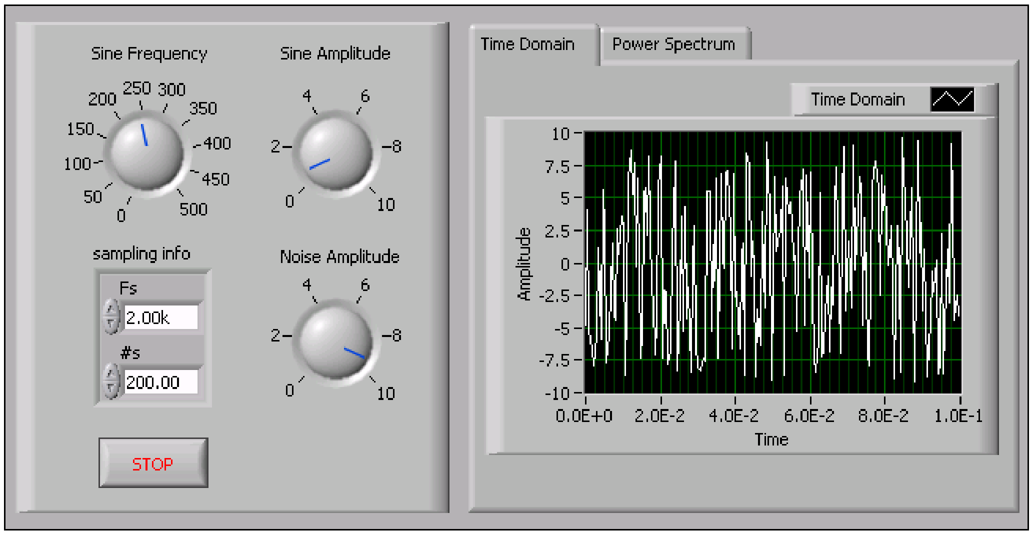
Exercise 7-2 LabVIEW Web Server

Objective: To use the LabVIEW Web Server tools to display a front panel in a Web browser.

Open the Enhanced Acquire Data VI you built in Exercise 2-4, and save its front panel into an HTML document.

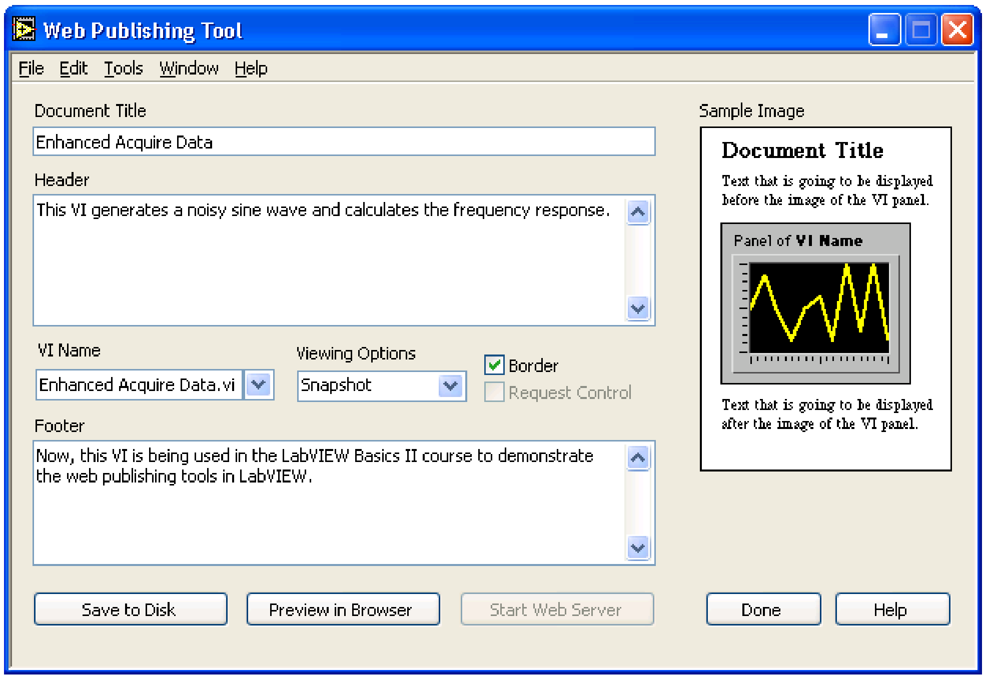
Front Panel

1. Open the Enhanced Acquire Data VI located in the `C:\Exercises\LabVIEW Basics II` directory.



2. Enable and configure the Web Server.
   a. Select **Tools»Options** and select **Web Server: Configuration** from the top pull-down menu.
   b. Place a checkmark in the **Enable Web Server** checkbox.
   c. Click the **OK** button to close the **Options** dialog box. The LabVIEW Web Server is now running.
3. Run the VI for a few seconds, then stop it.
4. Select **Tools»Web Publishing Tool** to display the **Web Publishing Tool** dialog box. Click the **Help** button to read more information on how to use this tool.

5. Configure the **Web Publishing Tool** similar to the following dialog box.


Web Publishing Tool
File Edit Tools Window Help
Sample Image
Document Title
Text that is going to be displayed before the image of the VI panel.
Panel of VI Name
Text that is going to be displayed after the image of the VI panel.
Save to Disk
Preview in Browser
Start Web Server
Done
Help

Document Title: Enhanced Acquire Data

Header: This VI generates a noisy sine wave and calculates the frequency response.

VI Name: Enhanced Acquire Data.vi

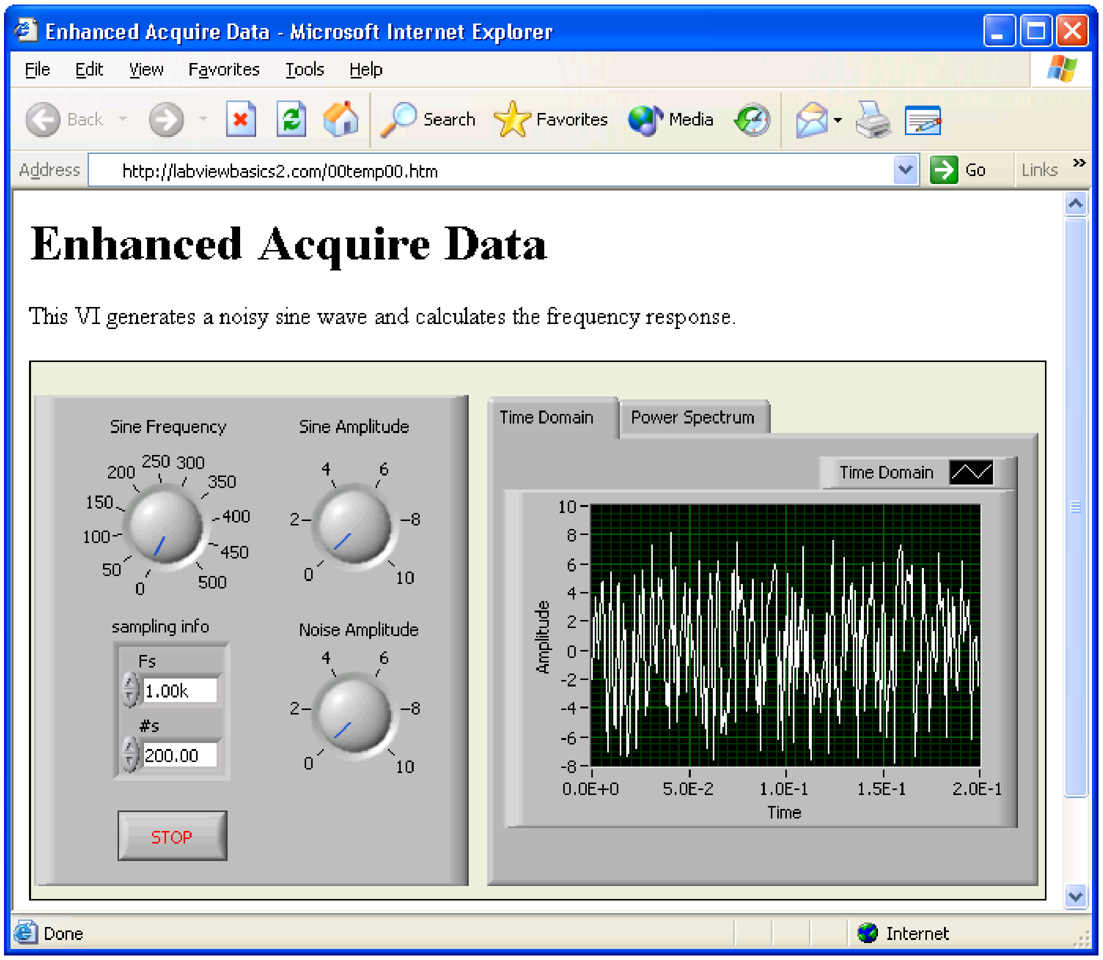
Viewing Options: Snapshot

Border [✓]

Request Control [ ]

Footer: Now, this VI is being used in the LabVIEW Basics II course to demonstrate the web publishing tools in LabVIEW.

6. Click the **Preview in Browser** button to open and display the front panel in a web browser. A window similar to the following example appears.


Enhanced Acquire Data - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://labviewbasics2.com/00temp00.htm Go Links
Enhanced Acquire Data
This VI generates a noisy sine wave and calculates the frequency response.
Sine Frequency
Sine Amplitude
sampling info
Fs 1.00k
#s 200.00
Noise Amplitude
STOP
Time Domain
Power Spectrum
Amplitude
Time
Done
Internet

Note The Monitor option works only with Netscape Navigator. You can view only static images using Internet Explorer.

7. Return to the **Web Publishing Tool** window and click the **Save to Disk** button to save the title, text, and VI front panel image in an HTML document. Save the document as `mypage.htm` in the `labview\www` directory.
8. Click the **Preview in Browser** button to open the HTML file that you saved in step 7 in a browser. Close the browser when you are finished.
9. Click the **Done** button to exit the Web Publishing Tool.
10. Close the Enhanced Acquire Data VI.

End of Exercise 7-2

C. Viewing and Controlling Front Panels Remotely

You can view and control a VI front panel remotely, either from within LabVIEW or from within a Web browser, by connecting to the LabVIEW built-in Web Server. When you open a front panel remotely from a client, the Web Server sends the front panel to the client, but the block diagram and all the subVIs remain on the server computer. You can interact with the front panel in the same way as if the VI were running on the client, except the block diagram executes on the server. Use this feature to publish entire front panels or to control your remote applications safely, easily, and quickly.

Note Use the LabVIEW Web Server if you want to control entire VIs. Use the DataSocket server to read and write data on a single front panel control in a VI. Refer to Chapter 17, *Networking in LabVIEW*, of the *LabVIEW User Manual* for more information about using the DataSocket Server.

Configuring the Server for Clients

The user at the server computer must first enable and configure the Web server before a client can view and control a front panel remotely using LabVIEW or a Web browser. Configure the Web Server by selecting **Tools» Options** and selecting the **Web Server** pages from the top pull-down menu. Use these pages to control browser access to the server and to specify which front panels are visible remotely. You also can use these pages to set a time limit on how long any particular remote client can control a VI.

The Web Server allows multiple clients to connect simultaneously to the same front panel, but only one client at a time can control the front panel. The user at the server computer can regain control of any VI at any time. When the controller changes a value on the front panel, all client front panels reflect that change. However, client front panels do not reflect all changes. In general, client front panels do not reflect changes made to the display on front panel objects, but rather to the actual values in the front panel objects. For example, if the controller changes the mapping mode or marker spacing of a scale of a chart or if the controller shows and hides a scrollbar for a chart, only the controller front panel reflects these changes.

Remote Front Panel License

You must configure a license to support the number of clients that potentially could connect to your server. By default, the remote front panel license included with LabVIEW allows one client to view and control a front panel. When you exceed the number of clients allowed by your license, the client receives a message that contains information you specify in the `LicenseErrorMessage.txt` file in `labview\www`. In this message, you might want to include information about who the client should contact, such as the server administrator or the person in your organization responsible for upgrading the remote front panel license. If you do not enter anything in this file, a default error message appears on the client computer indicating that the connection request was denied.

Note Although you can use any LabVIEW development system to build VIs that clients can view and control remotely, only the Full and Professional Development Systems support viewing and controlling those VIs remotely.

D. Viewing and Controlling Front Panels in LabVIEW

To view a remote front panel using LabVIEW as a client, open a new VI and select **Operate»Connect to Remote Panel** to display the **Connect to Remote Panel** dialog box. Use this dialog box to specify the server Internet address and the VI you want to view. By default, the remote VI front panel is initially in observer mode. You can request control by placing a checkmark in the **Request Control** checkbox in the **Connect to Remote Panel** dialog box when you request a VI. When the VI appears on your computer, right-click anywhere on the front panel and select **Request Control** from the shortcut menu. You also can access this menu by clicking the status bar at the bottom of the front panel window. If no other client is currently in control, a message appears indicating that you have control of the front panel. If another client is currently controlling the VI, the server queues your request until the other client relinquishes control. Only the user at the server computer can monitor the client queue list by selecting **Tools» Remote Panel Connection Manager**.

All VIs you want clients to view and control must be in memory. If the requested VI is in memory, the server sends the front panel of the VI to the requesting client. If the VI is not in memory, the **Connection Status** section of the **Open Remote Panel** dialog box displays an error message.

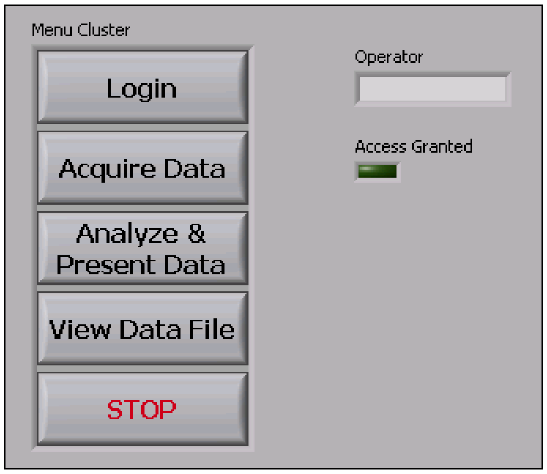
Exercise 7-3 Remote Front Panel VI

Objective: To view and control a VI from another computer using LabVIEW.

This exercise demonstrates how you can view and control other VIs remotely.

Front Panel

1. Open the Completed Application VI that you completed in Exercise 5-4.



Configuring the Web Server

2. Select **Tools»Options** and select **Web Server: Configuration** from the top pull-down menu.
3. Verify that the **Enable Web Server** checkbox is selected.
4. Select **Web Server: Browser Access** from the top pull-down menu and verify that the **Allow Viewing and Controlling** checkbox is selected.
5. Select **Web Server: Visible VIs** from the top pull-down menu and verify that the **Allow Access** checkbox is selected.
6. Click the **OK** button to close the **Options** dialog box.
7. Identify your IP address.

Note The following steps for identifying your IP address are specific to Windows XP. On other operating systems, refer to instructor or network administrator for more information about identifying your IP address.

a. From the desktop, select **Start»Settings»Control Panel** and double-click **Network Connections** to display the **Network Connections** window.

b. Right-click **Local Area Connection** and select **Properties** from the shortcut menu to display the **Local Area Connection Properties** dialog box.

c. On the **General** tab, select **Internet Protocol (TCP/IP)**. Click **Properties** to display the **TCP/IP Properties** dialog box.

d. If an IP address is listed, you have a static IP address. Record your IP address at the end of step 7. You will need this address later when communicating with other computers. For the hands-on course, the computers are not networked to a server and communicate peer-to-peer through LabVIEW. Close the **TCP/IP Properties** dialog box.

e. Close the **Local Area Connection Status** dialog box and the **Network Connections** window.

f. You also can obtain your IP address through the MS/DOS interface.

g. Select **Start»Programs»Accessories»Command Prompt** to display the **Command Prompt** window.

h. Type `ipconfig` at the prompt and press the <Enter> key.

i. Record your IP address if necessary.

j. Type `exit` at the prompt and press the <Enter> key to close the command prompt.

Tip If you have a dynamically obtained IP address, you can use the command prompt to obtain your current IP address. However, the IP address may change the next time you restart your computer. On some systems, the IP address can even change while it is still connected. Refer to your instructor or network administrator for further assistance on dynamically addressed systems.

IP address: ________________________________

8. Return to LabVIEW and run the Completed Application VI.

Note If you are performing this exercise on your own, perform the following steps on another computer that has the same version of LabVIEW.

Remotely View and Control the VI

9. Select **Operate»Connect to Remote Panel**.

10. In the dialog box that appears, type the IP address for your neighbor's computer in the **IP Address** field, type `Completed Application.vi` in the **VI Name** field, place a checkmark in the **Request Control** checkbox, and click the **Connect** button to exit the dialog box.

    (Optional) Instead of using an IP address, you can type the machine name in the **IP Address** field.

11. The Completed Application VI appears.
    - The message **`Control Granted`** appears on the remote front panel to indicate that the remote panel has control of the VI on the server machine.
    - The message **`Control granted to`** ***`machine name`***, where ***`machine name`*** is the name of the remote machine, appears on the server machine to indicate that a remote machine has control of the VI. You can click anywhere on the front panel of either VI to hide the message.
12. Experiment with the different settings on this VI. Notice that you are controlling your neighbor's VI.
13. On the server machine, right-click the front panel and select **Regain Control** from the shortcut menu. Control returns to the server machine.

    To prevent remote machines from gaining control of the VI, right-click the front panel and select **Remote Panel Server»Lock Control** from the shortcut menu. To allow remote control, right-click the front panel again and select **Remote Panel Server»Unlock Control** from the shortcut menu.
14. On the server machine, stop the VI. Notice the remote VI stops as well. Do not close the VI because you use it in the next exercise.

End of Exercise 7-3

E. Viewing and Controlling Front Panels from a Web Browser

If you want clients who do not have LabVIEW installed to be able to view and control a front panel remotely using a Web browser, they must install the LabVIEW Run-Time Engine. The LabVIEW Run-Time Engine includes a LabVIEW browser plug-in package that installs in the browser plug-in directory. The LabVIEW CD contains an installer for the LabVIEW Run-Time Engine.

Clients install the LabVIEW run-time engine and the user at the server computer creates an HTML file that includes an `<OBJECT>` tag that references the VI you want clients to view and control. This tag contains a URL reference to a VI and information that directs the Web browser to pass the VI to the LabVIEW browser plug-in. Clients navigate to the Web Server by entering the Web address of the Web Server in the address or URL field at the top of the Web browser window. The plug-in displays the front panel in the Web browser window and communicates with the Web Server so the client can interact with the remote front panel. Clients request control by selecting **Request Control of the VI** at the bottom of the remote front panel window in their Web browser or by right-clicking anywhere on the front panel and selecting **Request Control of the VI** from the shortcut menu.

Note National Instruments recommends that you use Netscape 4.7 or later or Internet Explorer 5.5 Service Pack 2 or later when viewing and controlling front panels in a Web browser.

Caveats for Viewing and Controlling Remote Front Panels

Because of the constraints of a Web browser, user interface applications that attempt to manipulate the dimensions and location of a front panel do not work properly when that front panel is displayed as a part of a Web page. Although the Web Server and the LabVIEW browser plug-in attempt to preserve the fidelity of complex user interface applications—in particular, those that present dialog boxes and subVI windows—some applications might not work properly in the context of a Web browser. National Instruments recommends that you do not export these types of applications for use in a Web browser.

In general, avoid exporting data-intensive VIs for remote viewing and controlling. For example, front panels that have several charts increase network load, which causes front panel updates to be slow. Also, VIs that have While Loops but no wait function limit background tasks from performing in a reasonable amount of time, making front panels unresponsive when viewed or controlled remotely.

Additionally, some VIs might not work exactly the same way from a remote computer as they do when run locally. ActiveX controls embedded on a front panel do not display on a remote client because they draw and operate almost completely independent of LabVIEW. If a VI presents the standard file dialog box, the controller receives an error because you cannot browse a file system remotely. Also, the **Browse** button of a path control is disabled in remote panels.

Clients viewing a front panel remotely might see different behavior depending on whether the front panel they are connecting to is from a built application. Specifically, if the front panel is from a built application, any programmatic changes to the front panel made before the client connects to the front panel are not reflected on the client computer. For example, if a property node changes a caption on a control before a client connects to that front panel, the client sees the original caption of the control, not the changed caption.

Block diagrams that achieve certain user interface effects by polling properties of a front panel control might experience decreases in performance when you control the VI from a remote computer. You can improve the performance of these VIs by using the Wait for Front Panel Activity function.

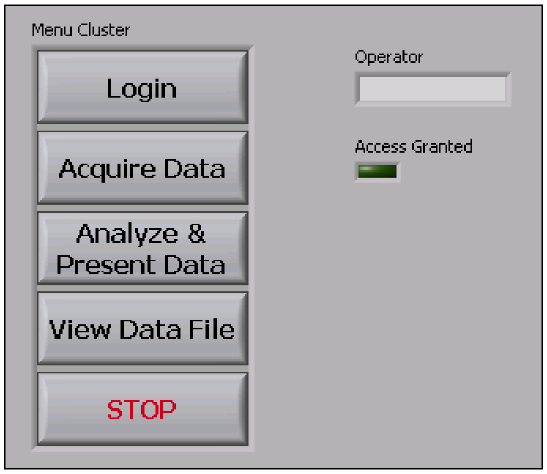
Exercise 7-4 Remote Panel VI

Objective: To view and control a front panel remotely from a Web browser.

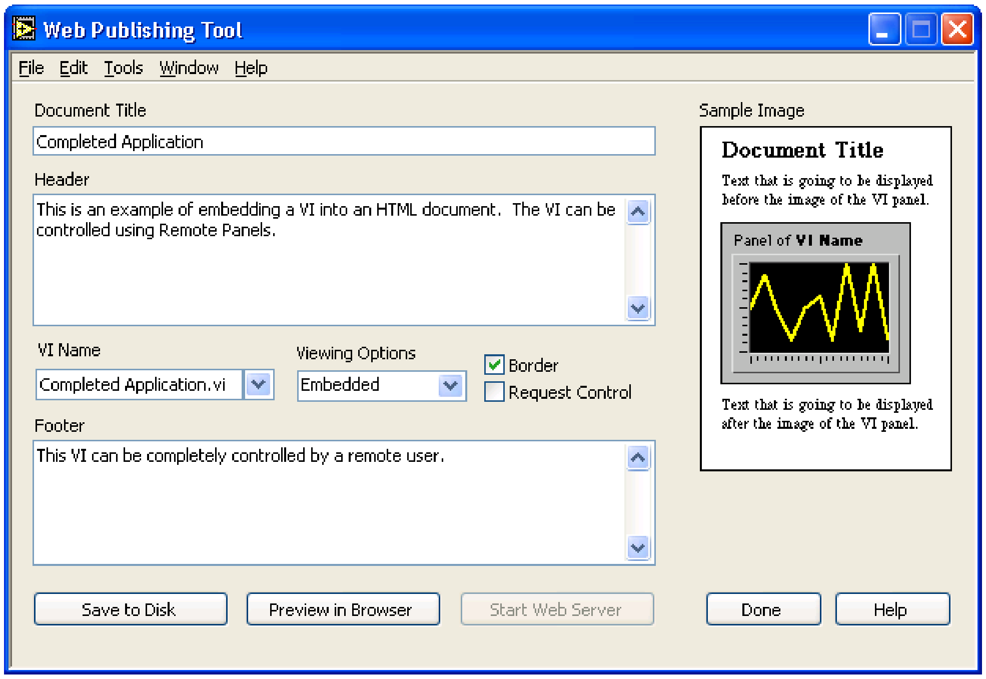
In this exercise, you view and control the front panel of the Completed Application VI from a Web browser.

Front Panel

1. Open the Completed Application VI in `C:\Exercises\LabVIEW Basics II` if you closed it at the end of the previous exercise.



2. Select **Tools»Web Publishing Tool**.
3. Configure the Web Publishing Tool similar to the following dialog box to create an HTML document that can embed a VI.


Web Publishing Tool
File Edit Tools Window Help
Save to Disk
Preview in Browser
Start Web Server
Done
Help
Sample Image
Document Title
Text that is going to be displayed before the image of the VI panel.
Panel of VI Name
Text that is going to be displayed after the image of the VI panel.

Document Title: Completed Application

Header: This is an example of embedding a VI into an HTML document. The VI can be controlled using Remote Panels.

VI Name: Completed Application.vi

Viewing Options: Embedded

[✓ Border] Request Control

Footer: This VI can be completely controlled by a remote user.

4. Click the **Save to Disk** button and save the document as `Basics2.html` in the `labview\www` directory.
5. On another machine, open a Web browser and enter `http://`*`machine name`*`/Basics2.html` in the address field, where *`machine name`* is the machine name or IP address of the server computer running the VI. The front panel of the Completed Application VI appears in the browser.
6. Right-click the front panel in the Web browser and select **Request Control of VI** from the shortcut menu. Experiment with different settings on the front panel.
7. From the Web browser, click the **STOP** button. Now, restart the VI. Notice you have complete control over the VI, including the ability to start and stop the VI.
8. Stop and close the VI.

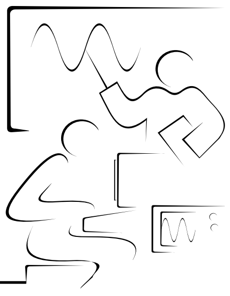
End of Exercise 7-4

Summary, Tips, and Tricks

- DataSocket is an Internet-based method of transferring data that is platform-independent, programming language-independent, and protocol-independent. DataSocket uses URLs to specify the specific data connection.
- DataSocket consists of two parts—the DataSocket API and the DataSocket Server.
- The DataSocket API for LabVIEW consists of the **Functions»All Functions»Communication»DataSocket** palette. The two main functions are DataSocket Write and DataSocket Read.
- You can have any control or indicator publish and/or subscribe data through the DataSocket Server by right-clicking that front panel object and using the **Data Operations»DataSocket Connection** window.
- You can view and control a VI front panel remotely, either from within LabVIEW or from within a Web browser, by connecting to the LabVIEW built-in Web Server.
- Use the LabVIEW Web Server to create HTML documents and publish front panel images on the Web. You can control browser access to the published front panels and configure which VIs are visible on the Web.

Notes

Appendix A
Additional Resources



This appendix contains the following sections of useful information for LabVIEW users:

A. Polymorphic VIs

B. Custom Graphics in LabVIEW

C. Additional Information

D. ASCII Character Code Equivalents Table

A. Polymorphic VIs

Polymorphic VIs accept different data types for a single input or output terminal. A polymorphic VI is a collection of subVIs with the same connector pane patterns. Each subVI is an instance of the polymorphic VI.

For example, the Read Key VI is polymorphic. Its **default value** terminal accepts Boolean, double-precision floating-point numeric, signed 32-bit integer numeric, path, string, or unsigned 32-bit integer numeric data.

The data types you wire to the inputs of a polymorphic VI determine the instance to use. If the polymorphic VI does not contain a subVI for that data type, a broken wire appears. You can select the instance you want to act as the default instance by right-clicking the polymorphic VI, selecting **Select Type** from the shortcut menu, and selecting the subVI.

Build your own polymorphic VIs when you perform the same operation on different data types.

Use polymorphic VIs to present a much simpler interface to the users of your VIs. Consider the case of a polymorphic VI that can sort either 1D or 2D arrays. Instead of having one VI for sorting 1D arrays and another subVI for sorting 2D arrays, one VI called Sort Array handles both types of inputs.

To create your own polymorphic VI, create two or more VIs with the same connector pane pattern. Select **File»New** and select **Polymorphic VI** in the **New** dialog box. In the dialog box that appears, add each VI to the polymorphic VI using the **Add VI** button.

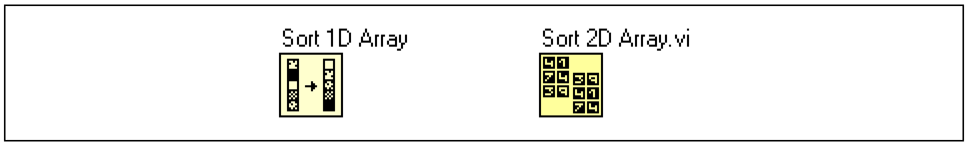
You can create an icon for the polymorphic VI using the **Edit Icon** button. You also can create context help for the polymorphic VI by selecting **Documentation** from the top pull-down menu in the **File»VI Properties** dialog box.

Note Context help for a polymorphic VI is not associated with context help for the VIs that compose the polymorphic VI. Therefore, you must create context help for the polymorphic VI.

Exercise A-1 Sort Poly Array VI

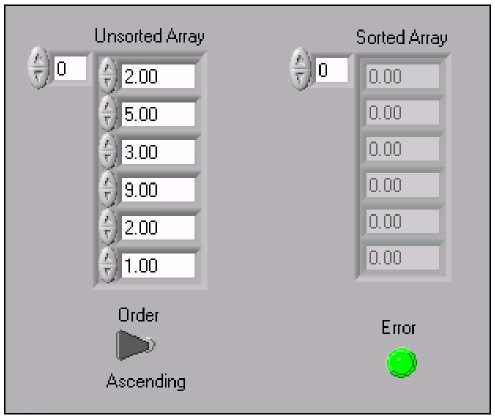
Objective: To create a polymorphic VI and use it as a subVI.

Using polymorphic VIs allows you to present a much simpler interface to the users of your VIs. Consider the case of a polymorphic VI that can sort either 1D or 2D arrays. Two individual VIs can do this function.

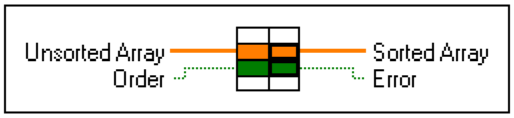


Instead of having one VI for sorting 1D arrays and another subVI for sorting 2D arrays, one VI called Sort Array handles both types of inputs. In this exercise, you create this VI.

1. Create a VI that sorts a 1D array. This VI should also include an option to sort in ascending or descending order and pass out an error Boolean object. The following example is a suggested front panel. You need to create the front panel and block diagram and test the code you write.



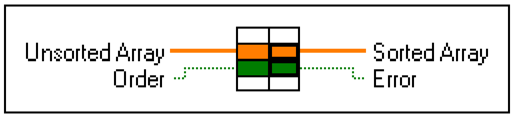
2. Create an icon connector pane using the following configuration.



Note Use the same icon connector pane for the Sort 2D Array instance. Otherwise, the polymorphic VI produces broken wires.

3. Save as `Sort 1D Array+.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

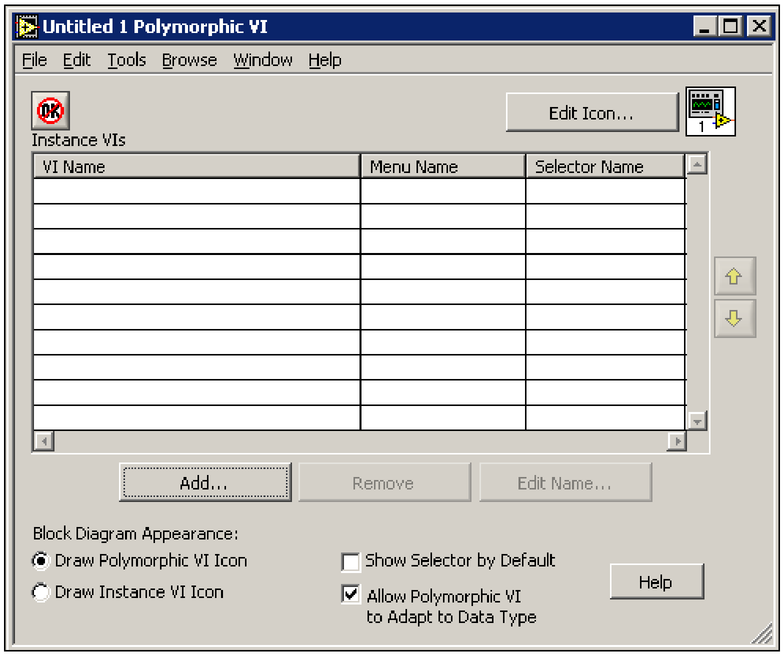
4. The Sort 2D Array function is already built for you. However, you need to complete the icon connector pane. This example was downloaded from `ni.com\support` in the Example Programs Database. Open this file from `C:\Exercises\LabVIEW Basics II\Sort 2D Array.vi`.
5. Create and wire the icon connector pane for the Sort 2D Array VI. Even though the icon connector pane is already built, you might want to modify it to help differentiate it from the Example program. You do not need to wire all the controls and indicators on this front panel. Recall the configuration used for the Sort 1D Array VI.



6. Save this new VI as `Sort 2D Array+.vi` in the `C:\Exercises\LabVIEW Basics II` directory.

Now create the polymorphic VI, which is composed of the Sort 1D Array+ instance and the Sort 2D Array+ instance.

7. To combine the two instances into one polymorphic VI, select **File»New** and select **Other Document Types»Polymorphic VI** from the **Create new** list.



Block Diagram Appearance:
(● Draw Polymorphic VI Icon) ○ Draw Instance VI Icon
☐ Show Selector by Default
☑ Allow Polymorphic VI to Adapt to Data Type

8. Add the Sort 1D Array+ VI and the Sort 2D Array+ VI using the **Add VI** button. You might need to browse to the directories where you saved these VIs.
9. Create an icon for this new polymorphic VI by selecting the **Edit Icon** button.
10. Create context help for this VI by selecting **Documentation** from the top pull-down menu in the **File»VI Properties** dialog box.
11. Save the VI as `Sort Poly Array.vi` in the `C:\Exercises\LabVIEW Basics II` directory.
12. Use this VI as a subVI in another VI to test the functionality. Notice the help screen. Also notice what happens when you double-click the polymorphic VI. What happens if you select a particular context by right-clicking and selecting **Select Type**?

End of Exercise A-1

B. Custom Graphics in LabVIEW

There are several LabVIEW features available for giving front panels a more professional, custom look. These features, provided with the LabVIEW full and professional development system, provide custom graphics and animation features to the user interface.

Decorations

One of the most straightforward methods to enhance a user interface is to apply the LabVIEW Decorations to a front panel as you did in Lesson 2, *VI Design Techniques*. Through careful use of the decorations, you can increase the readability of the front panels.

Importing Graphics

You can import graphics from other applications to use as front panel backgrounds, items in ring controls, and parts of other controls and indicators. Refer to the *LabVIEW Custom Controls, Indicators, and Type Definitions* Application Note for more information about using graphics in controls. LabVIEW supports most standard graphic formats, including animated GIF, MNG, animated MNG, and PNG. LabVIEW also supports transparency. Use one of the following methods to import a graphic.

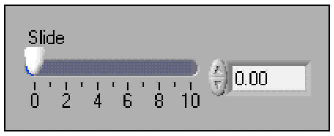
Note If you import an image by copying and pasting it, the image loses any transparency.

- **(Windows)** In the graphics editing application or Web browser, copy an image to the clipboard and switch to LabVIEW. The image is automatically available on the LabVIEW clipboard. Select **Edit»Paste** to place the image in LabVIEW. You also can select **Edit»Import Picture from File** or drag a graphics file from Windows Explorer.
- **(Macintosh)** In the graphics editing application or Web browser, copy an image to the clipboard and switch to LabVIEW. The image is automatically available on the LabVIEW clipboard. Select **Edit»Paste** to place the image in LabVIEW. You also can drag a closed graphic file from a folder and place it in LabVIEW.
- **(UNIX)** Select **Edit»Import Picture from File** to import a picture of type X Window System Dump (XWD), which you can create using the `xwd` command. Select **Edit»Paste** to place the image in LabVIEW.

Exercise A-2 Custom Slider Control

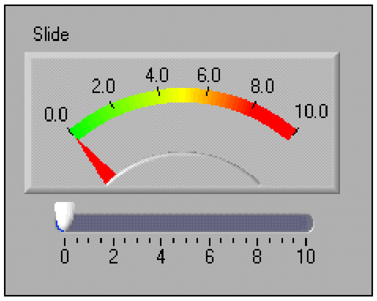
Objective: To use the Control Editor to modify a control.

1. Open a new front panel.
2. Place a Horizontal Pointer Slide, located on the **Controls»Numeric Controls** palette, on the front panel. Right-click the slide and select **Visible Items»Digital Display**.

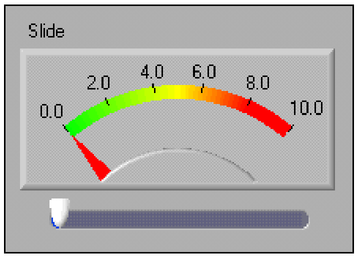


Modifying the Control

3. Launch the Control Editor by selecting the slide with the Positioning tool and selecting **Edit»Customize Control**. Using the Operating tool, move the slide to the middle of the front panel to allow more work space.
4. Right-click the digital display and select **Replace»Numeric»Meter**. Position the meter above the slide, as shown in the following example.



5. Hide the slide scale by right-clicking the slide and selecting **Scale» Style»None**.



6. Close the Control Editor by selecting **Close** from the **File** menu. Save the control as `Custom Slider.ctl` in the `C:\Exercises\LabVIEW Basics II` directory, then click **Yes** to replace the existing one. The modified slider is shown on the front panel.

Note You can save controls that you create like you save VIs. You can load saved controls using **Select a Control** from the **Controls** palette. Controls have a `.ctl` extension.

7. Manipulate the slider and watch the meter track its data value.
8. Close the VI. Do not save changes.

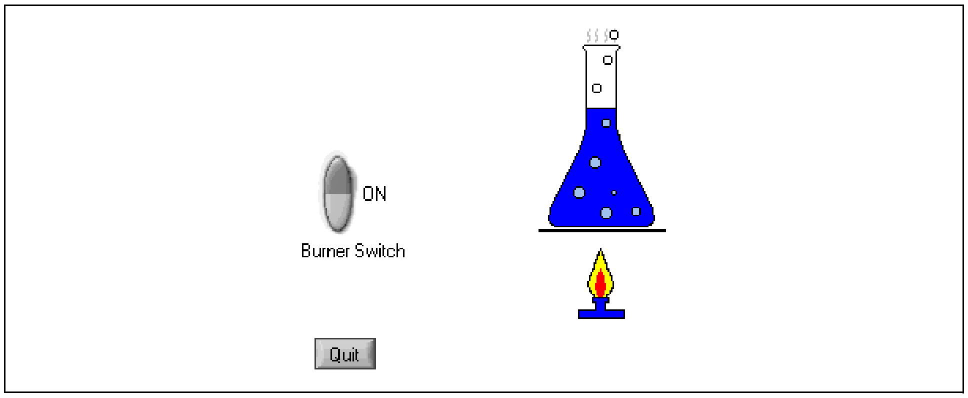
End of Exercise A-2

Exercise A-3 Custom Picture Exercise VI

Objective: To create a custom Boolean indicator.

Build a VI that uses custom Boolean indicators to show the state of a Bunsen burner and flask being heated. The pictures representing the on and off states of the Bunsen burner and the flask are already drawn for you.

Front Panel



1. Open the Custom Picture Exercise VI located in the `C:\Exercises\LabVIEW Basics II` directory.

   The VI contains a vertical rocker switch to turn the Bunsen burner on and off, and a button to quit the application. It also contains two graphics representing the on and off states of the Bunsen burner, and two graphics representing the boiling and non-boiling states of the flask.

2. To create the custom flask Boolean object, complete the following steps.

   a. Right-click an open area on the front panel and select **Square LED** from the **Controls»LEDs** palette. Label the LED `Flask`.

   b. Using the Positioning tool, select the graphic that shows the contents of the flask boiling and select **Edit»Cut**. Click the Flask LED indicator and select **Edit»Customize Control**. The Control Editor now appears with the Flask LED displayed. Right-click the LED and select **Import Picture»True**. This custom picture now represents the TRUE state.

Note The default state of the LED is FALSE. If you do not see the picture, the LED is probably in the FALSE state.

   c. Change to the front panel by clicking it. Using the Positioning tool, select the graphic of the flask that shows the contents of the flask not boiling, and select **Edit»Cut**. Change to the **Control Editor** window by clicking it.

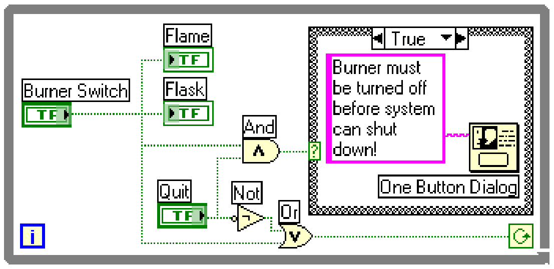
d. Right-click the boiling flask and select **Import Picture»False**. This custom picture now represents the FALSE state.

e. Select **Apply Changes** from the **File** menu, and close the Control Editor. Do not save the custom control.

3. Right-click an open area and select **Square LED** from the **Controls»All Controls»Classic Controls»Boolean** shortcut menu. Label the LED `Flame`.
4. Using the previous steps, make the LED look like a Bunsen burner. The TRUE state should show the burner on; the FALSE state should show the burner off.
5. Hide the labels of both Boolean indicators by right-clicking them and selecting **Visible Items»Label**. Select both Boolean indicators and align them on horizontal centers using the **Align Objects** tool.

Block Diagram

6. Complete the following block diagram.



7. Save the VI under the same name.
8. Return to the front panel and run the VI. Turn the Burner Switch on and off and notice the custom Boolean change.
9. Stop the VI by clicking the **Quit** button. If you click the **Quit** button while the burner is on, a dialog box notifies you that the burner must be off before you can shut down the system.
10. Close the VI when you are finished.

End of Exercise A-3

C. Additional Information

This section describes how you can receive more information regarding LabVIEW, instrument drivers, and other topics related to this course.

National Instruments Technical Support Options

The best resource for getting technical support and other information about LabVIEW, test and measurement, instrumentation, and other National Instruments products and services is the NI Web site at `ni.com`.

The support page for the National Instruments Web site contains links to application notes, the support KnowledgeBase, hundreds of examples, and troubleshooting wizards for all topics discussed in this course and more.

Another excellent place to obtain support while developing various applications with National Instruments products is the NI Developer Zone at `ni.com/zone`.

The NI Developer Zone also includes direct links to the instrument driver network and to Alliance Program member Web pages.

The Alliance Program

The National Instruments Alliance Program joins system integrators, consultants, and hardware vendors to provide comprehensive service and expertise to customers. The program ensures qualified, specialized assistance for application and system development. Information about and links to many of the Alliance Program members are available from the National Instruments Web site.

User Support Newsgroups

The National Instruments User Support Newsgroups are a collection of Usenet newsgroups covering National Instruments products as well as general fields of science and engineering. You can read, search, and post to the newsgroups to share solutions and find additional support from other users. You can access the User Support Newsgroups from the National Instruments support Web page.

Other National Instruments Training Courses

National Instruments offers several training courses for LabVIEW users. The courses are listed in the National Instruments catalog and online at `ni.com/training`. These courses continue the training you received here and expand it to other areas. You can purchase just the course materials or sign up for an instructor-led hands-on course by contacting National Instruments.

LabVIEW Publications

LabVIEW Technical Resource (LTR) Newsletter

Subscribe to *LabVIEW Technical Resource* to discover power tips and techniques for developing LabVIEW applications. This quarterly publication offers detailed technical information for novice users as well as advanced users. In addition, every issue contains a disk of LabVIEW VIs and utilities that implement methods covered in that issue. To order *LabVIEW Technical Resource*, call LTR publishing at (214) 706-0587 or visit `ltrpub.com`.

LabVIEW Books

Many books have been written about LabVIEW programming and applications. The National Instruments Web site contains a list of all the LabVIEW books and links to places to purchase these books. Publisher information is also included so you can directly contact the publisher for more information on the contents and ordering information for LabVIEW and related computer-based measurement and automation books.

The Info-labview Listserve

`Info-labview` is an e-mail group of users from around the world who discuss LabVIEW issues. The people on this list can answer questions about building LabVIEW systems for particular applications, where to get instrument drivers or help with a device, and problems that appear.

Send subscription messages to the `info-labview` list processor at:

`listmanager@pica.army.mil`

Send other administrative messages to the `info-labview` list maintainer at:

`info-labview-REQUEST@pica.army.mil`

Post a message to subscribers at:

`info-labview@pica.army.mil`

You also might want to search the ftp archives at:

`ftp://ftp.pica.army.mil/pub/labview/`

The archives contain a large set of donated VIs for doing a wide variety of tasks.

D. ASCII Character Code Equivalents Table

The following table contains the hexadecimal, octal, and decimal code equivalents for ASCII character codes.

Hex	Octal	Decimal	ASCII	Hex	Octal	Decimal	ASCII
00	000	0	NUL	20	040	32	SP
01	001	1	SOH	21	041	33	!
02	002	2	STX	22	042	34	"
03	003	3	ETX	23	043	35	#
04	004	4	EOT	24	044	36	$
05	005	5	ENQ	25	045	37	%
06	006	6	ACK	26	046	38	&
07	007	7	BEL	27	047	39	'
08	010	8	BS	28	050	40	(
09	011	9	HT	29	051	41	)
0A	012	10	LF	2A	052	42	*
0B	013	11	VT	2B	053	43	+
0C	014	12	FF	2C	054	44	,
0D	015	13	CR	2D	055	45	-
0E	016	14	SO	2E	056	46	.
0F	017	15	SI	2F	057	47	/
10	020	16	DLE	30	060	48	0
11	021	17	DC1	31	061	49	1
12	022	18	DC2	32	062	50	2
13	023	19	DC3	33	063	51	3
14	024	20	DC4	34	064	52	4
15	025	21	NAK	35	065	53	5
16	026	22	SYN	36	066	54	6
17	027	23	ETB	37	067	55	7

Hex	Octal	Decimal	ASCII
18	030	24	CAN
19	031	25	EM
1A	032	26	SUB
1B	033	27	ESC
1C	034	28	FS
1D	035	29	GS
1E	036	30	RS
1F	037	31	US
40	100	64	@
41	101	65	A
42	102	66	B
43	103	67	C
44	104	68	D
45	105	69	E
46	106	70	F
47	107	71	G
48	110	72	H
49	111	73	I
4A	112	74	J
4B	113	75	K
4C	114	76	L
4D	115	77	M
4E	116	78	N
4F	117	79	O
50	120	80	P
51	121	81	Q
52	122	82	R

Hex	Octal	Decimal	ASCII
38	070	56	8
39	071	57	9
3A	072	58	:
3B	073	59	;
3C	074	60	<
3D	075	61	=
3E	076	62	>
3F	077	63	?
60	140	96	`
61	141	97	a
62	142	98	b
63	143	99	c
64	144	100	d
65	145	101	e
66	146	102	f
67	147	103	g
68	150	104	h
69	151	105	i
6A	152	106	j
6B	153	107	k
6C	154	108	l
6D	155	109	m
6E	156	110	n
6F	157	111	o
70	160	112	p
71	161	113	q
72	162	114	r

Hex	Octal	Decimal	ASCII
53	123	83	S
54	124	84	T
55	125	85	U
56	126	86	V
57	127	87	W
58	130	88	X
59	131	89	Y
5A	132	90	Z
5B	133	91	[
5C	134	92	\
5D	135	93	]
5E	136	94	^
5F	137	95	_

Hex	Octal	Decimal	ASCII
73	163	115	s
74	164	116	t
75	165	117	u
76	166	118	v
77	167	119	w
78	170	120	x
79	171	121	y
7A	172	122	z
7B	173	123	{
7C	174	124	|
7D	175	125	}
7E	176	126	~
7F	177	127	DEL

Notes

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: *LabVIEW Basics II: Development Course Manual*

Edition Date: June 2003

Part Number: 320629L-01

Please comment on the completeness, clarity, and organization of the manual.

____

____

____

____

____

____

____

____

If you find errors in the manual, please record the page numbers and describe the errors.

____

____

____

____

____

____

____

Date manual was purchased (month/year): ______________________________

Thank you for your help.

Name ____

Title ____

Company ____

Address ____

____

Email Address ____

Phone (___)_______________________ Fax (___) _______________________

Mail to: Customer Education
National Instruments Corporation
11500 North Mopac Expressway
Austin, Texas 78759-3504

Fax to: Customer Education
National Instruments Corporation
512 683 6837

Course Evaluation

Course ____

Location ____

Instructor ______________________ Date ______________________

Student Information (optional)

Name ____

Company ______________________ Phone ______________________

Instructor

Please evaluate the instructor by checking the appropriate circle.	Unsatisfactory	Poor	Satisfactory	Good	Excellent
Instructor's ability to communicate course concepts	❍	❍	❍	❍	❍
Instructor's knowledge of the subject matter	❍	❍	❍	❍	❍
Instructor's presentation skills	❍	❍	❍	❍	❍
Instructor's sensitivity to class needs	❍	❍	❍	❍	❍
Instructor's preparation for the class	❍	❍	❍	❍	❍

Course

Training facility quality	❍	❍	❍	❍	❍
Training equipment quality	❍	❍	❍	❍	❍

Was the hardware set up correctly? ❍ Yes ❍ No

The course length was ❍ Too long ❍ Just right ❍ Too short

The detail of topics covered in the course was ❍ Too much ❍ Just right ❍ Not enough

The course material was clear and easy to follow. ❍ Yes ❍ No ❍ Sometimes

Did the course cover material as advertised? ❍ Yes ❍ No

I had the skills or knowledge I needed to attend this course. ❍ Yes ❍ No If no, how could you have been better prepared for the course? ____

____

What were the strong points of the course? ____

____

What topics would you add to the course? ____

____

What part(s) of the course need to be condensed or removed? ____

____

What needs to be added to the course to make it better? ____

____

Are there others at your company who have training needs? Please list. ____

____

____

Do you have other training needs that we could assist you with? ____

____

How did you hear about this course? ❍ NI Web site ❍ NI Sales Representative ❍ Mailing ❍ Co-worker
❍ Other ____