

Course Software Version 6.0
September 2000 Edition
Part Number 320629G-01

Copyright

Copyright © 1993, 2000 by National Instruments Corporation, 11500 North Mopac Expressway, Austin, Texas 78759-3504. Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

LabVIEW™, National Instruments™, and ni.com™ are trademarks of National Instruments Corporation. Product and company names mentioned herein are trademarks or trade names of their respective companies.

Worldwide Technical Support and Product Information

ni.com

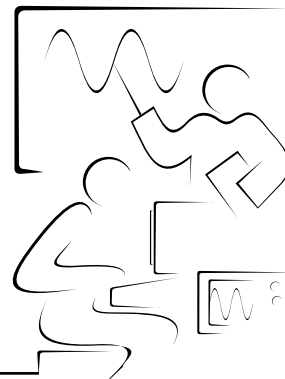
National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 794 0100

Worldwide Offices

Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 284 5011, Canada (Calgary) 403 274 9391, Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, China 0755 3904939, Denmark 45 76 26 00, Finland 09 725 725 11, France 01 48 14 24 24, Greece 30 1 42 96 427, Germany 089 741 31 30, Hong Kong 2645 3186, India 91805275406, Israel 03 6120092, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico (D.F.) 5 280 7625, Mexico (Monterrey) 8 357 7695, Netherlands 0348 433466, New Zealand 09 914 0488, Norway 32 27 73 00, Poland 0 22 528 94 06, Portugal 351 1 726 9011, Singapore 2265886, Spain 91 640 0085, Sweden 08 587 895 00, Switzerland 056 200 51 51, Taiwan 02 2528 7227, United Kingdom 01635 523545

Contents



Student Guide

A. About This Manual	SG-1
B. What You Need to Get Started	SG-3
C. Installing the Course Software.....	SG-4
D. Course Goals and Non-Goals	SG-5
E. Course Map.....	SG-6
F. Course Conventions.....	SG-7

Lesson 1

Planning LabVIEW Applications

A. The Planning and Design Process.....	1-2
B. The Implementation Process.....	1-3
C. Error Handling Techniques.....	1-4
D. LabVIEW Programming Architectures	1-10
E. VI Templates	1-21
Summary, Tips, and Tricks.....	1-24

Lesson 2

Designing Front Panels

A. Basic User Interface Issues	2-2
B. Using Boolean Clusters as Menus	2-14
C. Property Nodes	2-24
Common Properties	2-27
D. Graph and Chart Properties	2-37
E. Control References	2-46
F. LabVIEW Run-Time Menus (Optional).....	2-51
G. Intensity Plots	2-60
Summary, Tips, and Tricks.....	2-64
Additional Exercises	2-65

Lesson 3

Data Management Techniques

A. Data Management Techniques in LabVIEW	3-2
B. Local Variables	3-4
C. Global Variables	3-14
D. Important Advice about Local and Global Variables	3-23
E. DataSocket	3-26
Summary, Tips, and Tricks	3-35
Additional Exercises	3-36

Lesson 4

Advanced File I/O Techniques

A. Working with Byte Stream Files	4-2
B. LabVIEW Datalog Files	4-13
C. Streaming Data to Disk	4-20
Summary, Tips, and Tricks	4-21
Additional Exercises	4-22

Lesson 5

Developing Larger Projects in LabVIEW

A. Assembling a LabVIEW Application	5-2
B. LabVIEW Features for Project Development	5-13
C. LabVIEW Tools for Project Management	5-21
Summary, Tips, and Tricks	5-35
Additional Exercises	5-36

Lesson 6

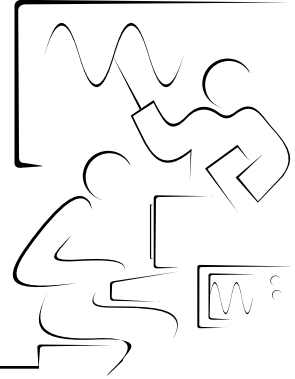
Performance Issues

A. LabVIEW Multithreading and Multitasking Overview	6-2
B. The Profile Window	6-6
C. Speeding Up Your VIs	6-12
D. System Memory Issues	6-25
E. Optimizing VI Memory Use	6-28
Summary, Tips, and Tricks	6-46

Appendix

A. Polymorphic SubVIs	A-2
B. Custom Graphics in LabVIEW	A-7
C. The LabVIEW Web Server	A-14
D. Additional Information	A-20
E. ASCII Character Code Equivalents Table	A-22

Student Guide



Thank you for purchasing the LabVIEW Basics II 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 course. You can apply the full purchase of this course kit towards the corresponding course registration fee if you register within 90 days of purchasing the kit. Visit the Customer Education section of ni.com for online course schedules, syllabi, training centers, and class registration.

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 course or that you 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.

If you do not have this hardware, you still can complete most of the exercises. Be sure to use the demo versions of the VIs when you are working through exercises. 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.



Each exercise shows a picture of a *finished* front panel and block diagram after you run the VI, as shown in the following illustration. After each block diagram picture is a description of each object in the block diagram.

①

②

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 95 or later installed on your computer; **(Macintosh)** Power Macintosh running MacOS 7.6.1 or later; **(UNIX)** Sun workstation running Solaris 2.5 or later and XWindows system software, an HP 9000 workstation model 700 series running HP-UX 10.20 or later, or a PC running Linux kernel 2.0.x or later for the Intel x86 architecture
- (Windows)** Multifunction DAQ device configured as device 1 using Measurement & Automation Explorer; **(Macintosh)** Multifunction DAQ device in Slot 1
- DAQ Signal Accessory, wires, and cable
- LabVIEW Professional Development System 6.0 or later
- (Optional) A word processing application such as **(Windows)** Notepad, WordPad, **(Macintosh)** TeachText, **(UNIX)** Text Editor, vi, or vuedpad
- LabVIEW Basics II course disk, containing the following files.

Filename	Description
LVB2SW.exe	Self-extracting archive containing VIs used in the course
LVB2Sol.exe	Self-extracting archive 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 LabVIEW Basics II course software.

Windows

1. Run the program called `LVB2SW.exe`. The course files will be extracted to the `c:\exercises\LV Basics 2` directory:
`Basics2.llb` will be installed in the `LabVIEW\user.lib` directory. When you launch LabVIEW, a palette called **Basics 2 Course** will be in the **User Libraries** palette of the **Functions** palette.
2. (Optional) Double-click `LVB2Sol.exe` to install the solutions to all exercises in the `c:\solutions\LV Basics 2` directory.

Macintosh

1. As shown in steps 1 and 2 of the Windows installation, use a Windows-based PC to extract the files and transfer them to your Macintosh. If you do not have access to a PC, contact National Instruments for uncompressed files.
2. Copy the files to your hard disk using the directory structure described in the [Windows](#) section.

UNIX

1. As shown in steps 1 and 2 of the Windows installation, use a Windows-based PC to extract the files and transfer them to your workstation. If you do not have access to a PC, contact National Instruments for uncompressed files.
2. Mount the PC disk you are using to transfer the files. The course assumes the directory structure described in the [Windows](#) section. Copy all files to the appropriate location.

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 your applications.
- Improve memory usage and performance of your VIs.

You will apply these concepts in Lesson 5, *Developing Larger Projects in LabVIEW*. In Lesson 5, you will build a project that uses VIs you create in Lessons 1, 2, 3, and 4. While these VIs individually illustrate specific concepts and features in LabVIEW, they constitute part of a larger project you will finish in Lesson 5.

The project you will build must meet the following criteria:

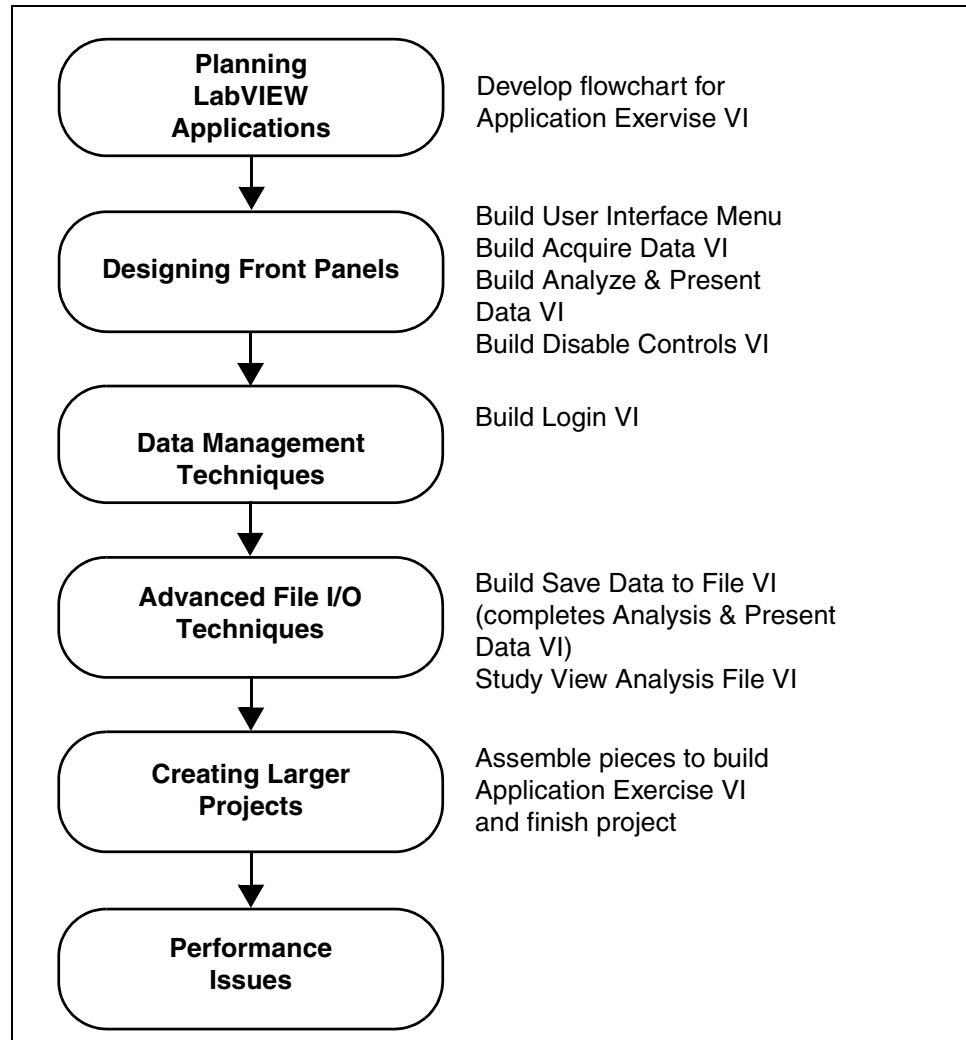
- Provides a menu-like user interface.
- Requires the user to log in with a correct name and password.
- If the user is not correctly logged in, other features are disabled.
- Acquires data with the specified user configuration.
- The user can analyze a subset of data and save the results to a file.
- The user can load and view analysis results previously saved to disk.

The following course map contains notes about the parts of the project you will develop in various sections of the course. Exercises within the lessons also remind you when you are working on a VI used in a later exercise.

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

- LabVIEW programming methods covered in the LabVIEW Basics I course
- Programming theory
- Every built-in VI, function, or object
- Developing a complete application for any student in the class


E. Course Map




F. 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 note, which alerts you to important information.

-  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.

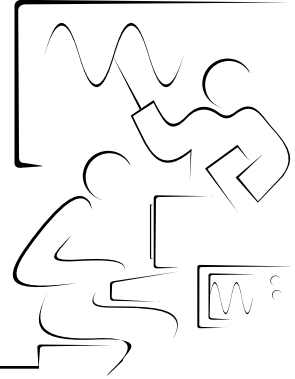
- monospace Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and code excerpts.

- Platform** Text in this font denotes a specific platform and indicates that the text following it applies only to that platform.

- right-click **(Macintosh)** Press <Command>-click to perform the same action as a right-click.

Lesson 1

Planning LabVIEW Applications



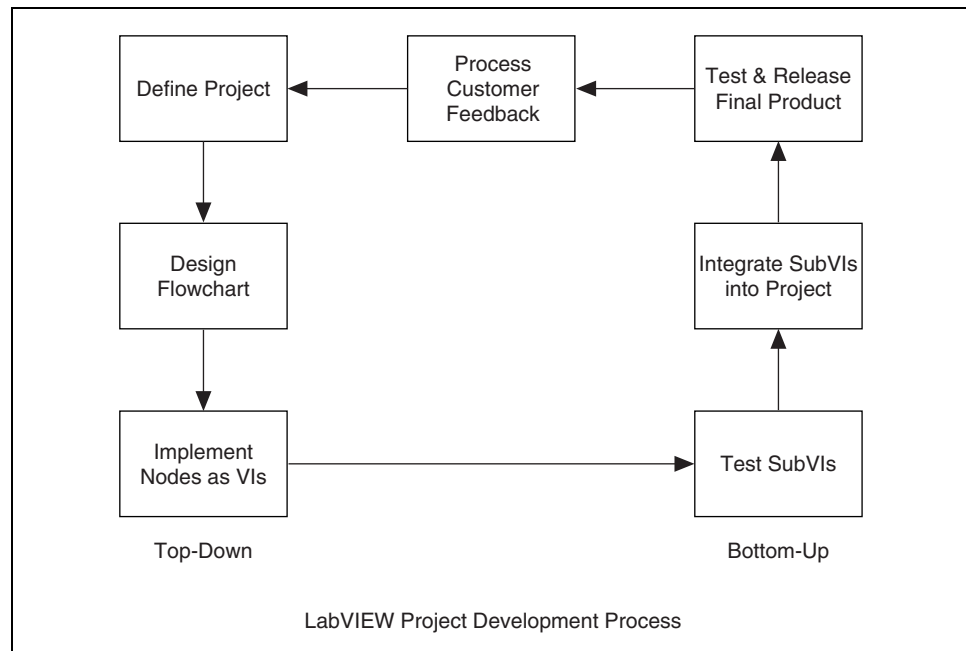
This lesson describes some of the issues involved when developing LabVIEW applications, including the design process, the organization of subVI components, and the process of combining those components to create a complete application. This lesson also describes common LabVIEW programming architectures along with some tools to help you build VIs.

You Will Learn:

- A. Planning and design tips for developing a LabVIEW application
- B. How to convert your design outline into actual LabVIEW subVIs
- C. Error handling techniques
- D. Common LabVIEW programming architectures
- E. About VI templates

A. The Planning and Design Process

To design large LabVIEW projects, you will find that you usually begin with a top-down approach. That is, you first define the general characteristics and specifications of the project. After you define the requirements of the application with input from your customer, you begin developing the subVIs you will eventually assemble to form the completed project. This stage 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 chart illustrates this project development process.



Designing a flow diagram can help you visualize how your application should operate and set up the overall hierarchy of your project. Because LabVIEW is a data flow programming language and its block diagrams are similar to typical flowcharts, it is important to carefully plan this chart. 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.

Also, keep in mind the following development guidelines:

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

B. The Implementation Process

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

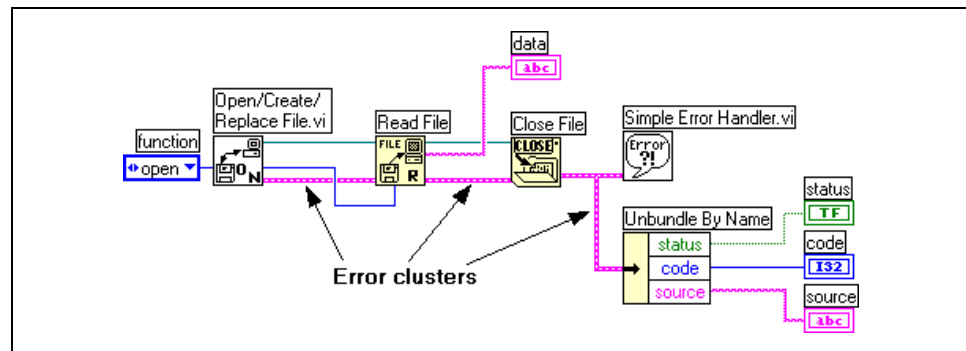
In addition, modularization makes it much easier for you to test small portions of an application and later combine them. If you build an entire application on one block diagram without subVIs, you might not be able to start testing until you have developed the majority of the application. At that point, it is very cumbersome to debug problems that might arise. Further, 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 keep abreast of the project's status. Another advantage of using subVIs is that future modifications and improvements to the application will be much easier to implement.

After you build and test the necessary subVIs, you will use them to complete your LabVIEW application. This is the bottom-up portion of the development.

C. Error Handling Techniques

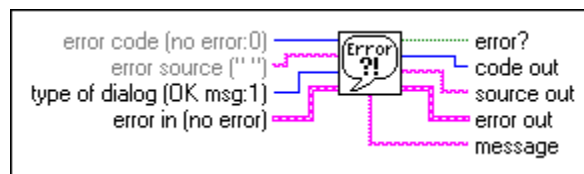
In the LabVIEW Basics I course, you used the error in and error out clusters to pass error information between functions and subVIs. These error clusters contain three pieces of information—a status Boolean indicating a True value for an error, a numeric that indicates the error number, and the source string that displays which function or subVI generated the error. You can use either the Unbundle or the Unbundle By Name function located on the **Functions»Cluster** palette to extract this information as shown in the following block diagram.



The previous example illustrates a typical usage of the error in/error out approach. That is, the File I/O VIs and functions use error clusters to pass information from one operation to the next. You can then use the error handling VIs from the bottom row of the **Time & Dialog** palette to notify the user of any error conditions that occur.

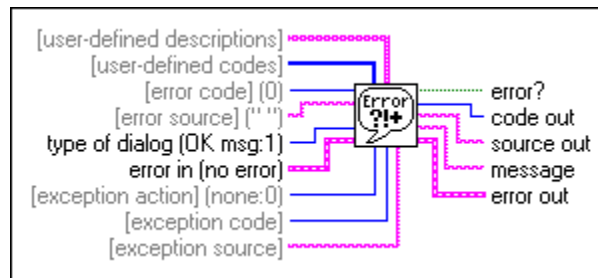
The Error Handler VIs

The Simple Error Handler takes the error in cluster or the error code value and if an error occurs, opens a dialog box that describes the error and possible reasons for it. You also can change the type of dialog box it opens from displaying an **OK** button, to not display a dialog box at all, or to display a dialog box and give the user a choice to continue or stop the VI.



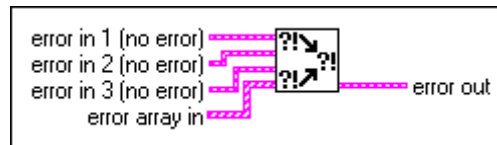
The General Error Handler VI also accepts the error in cluster or the error code value and a dialog box of the type specified appears when an error occurs. However, in addition, you can set up error exception lists so that specified errors are cleared or set when they occur. You also can use the General 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 how to modify your error handler VIs.



When an error occurs, the Simple Error Handler and General Error Handler VIs open a dialog box that displays the information contained in the error cluster and possible reasons for that error as listed in the internal error description table.

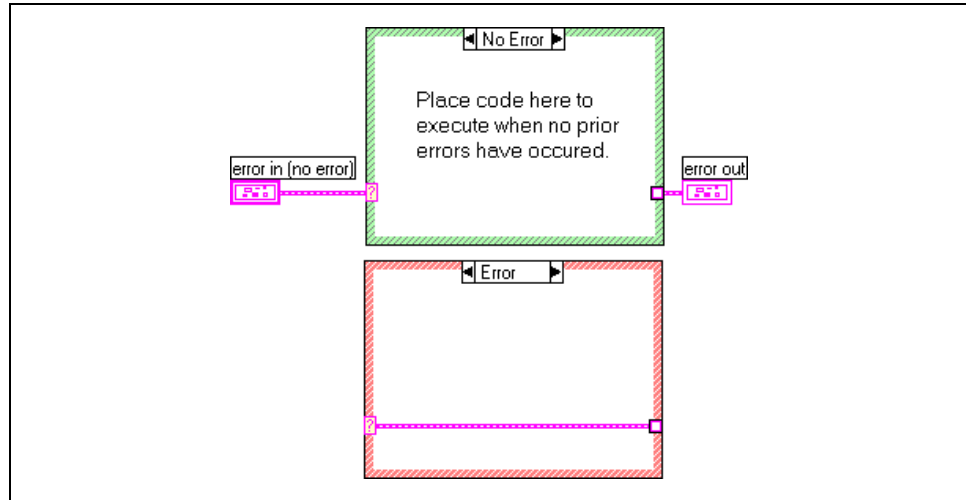
Sometimes you have separate lines of operations that run in parallel in LabVIEW and each operation maintains their own error clusters. You can use the Merge Errors VI to combine several error clusters into one.



The Merge Errors VI looks at the incoming error clusters or the array of error clusters and outputs the first error found. If no errors occur, LabVIEW returns the first warning message, error code is a positive value. Otherwise, LabVIEW returns a no error condition.

Incorporating Error Handling into Your VIs

You should build error handling into your own VIs in addition to using the error clusters for built-in VIs and functions. For example, when you are building a subVI to use in a larger project, you might not want that subVI to run if an error occurred previously. You can wire the error cluster to a Case structure to get Error and No Error cases as shown in the following example.



As shown in the previous example, you place the code you wish to run in the No Error case and then define the error out value for that case depending upon what is occurring in that case.

In the next exercise you will build a VI that generates data, analyzes those data, and presents the data to the front panel while using error clusters appropriately.

Exercise 1-1 Generate & Analyze Data VI

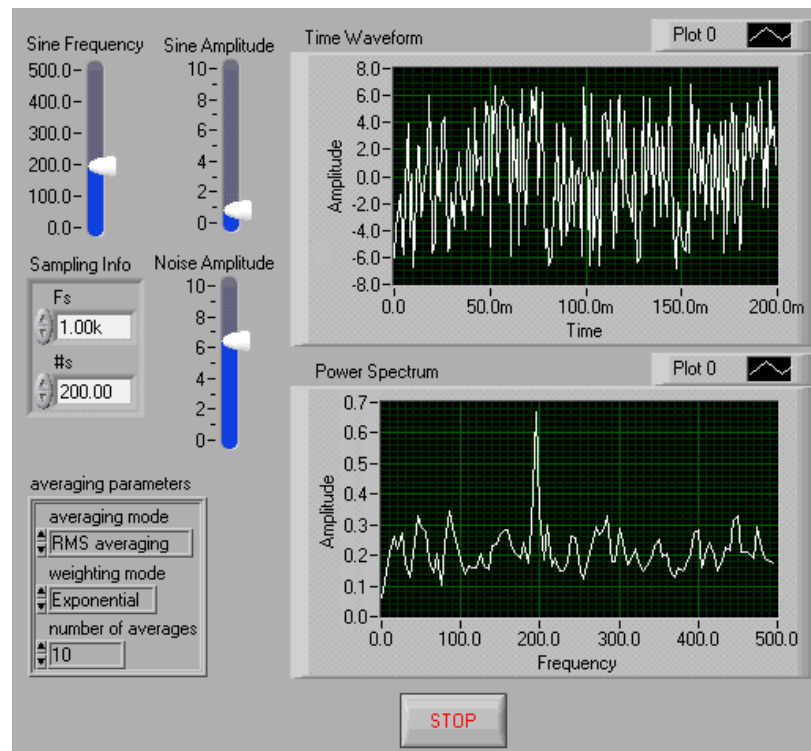
Objective: To build a VI that generates, analyzes, and displays data while using error handling techniques.

You will build a VI that generates a noisy sine waveform, computes the frequency response of those data, and plots the time and frequency waveforms in waveform graphs. You will use the error clusters and the error handling VIs to properly monitor error conditions.



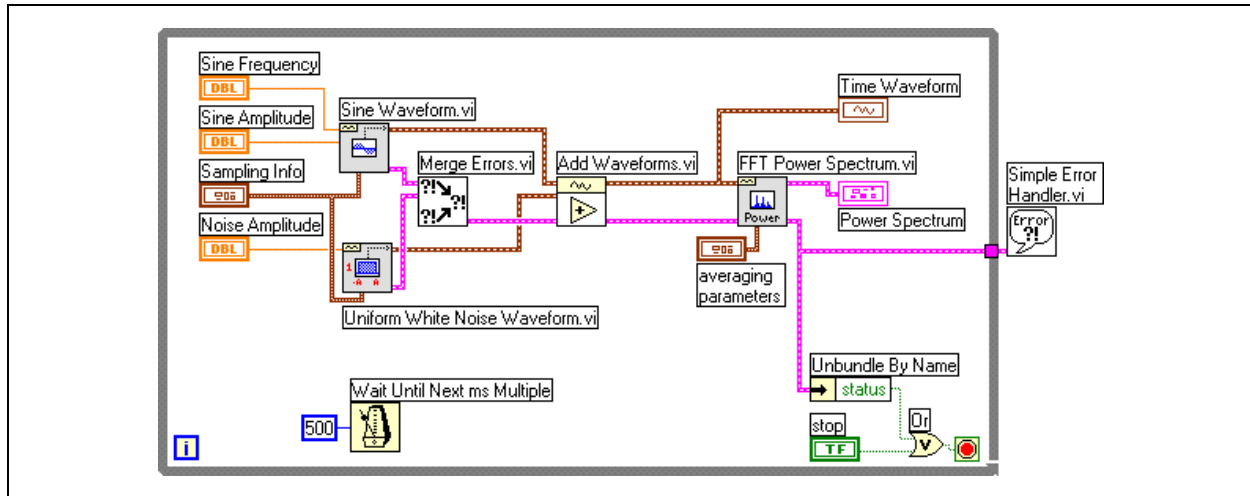
Note You will use this VI in the appendix.

Front Panel



1. Open a new VI.
2. Add the three Vertical Pointer Slides located on the **Controls»Numeric** palette, the **Stop** button located on the **Controls»Boolean** palette, and the two Waveform Graphs located on the **Controls»Graph** palette to the front panel as shown in the previous front panel. Label them appropriately. You will create the two clusters—Sampling Info and averaging parameters—from the block diagram.

Block Diagram



3. Open and build the block diagram using the following components.



a. Place a While Loop located on the **Functions»Structures** palette on the block diagram. This structures the VI to continue to generate and analyze data until the user clicks the **Stop** button. Right-click the Conditional terminal and select **Stop If True**.



b. Place the Sine Waveform VI located on the **Functions»Analyze»Waveform Generation** palette on the block diagram. This VI generates a sine waveform with the specified frequency, amplitude, and sampling information. To create the sampling info cluster control, right-click that input terminal and select **Create»Control** from the shortcut menu.



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



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



e. Place the Add Waveforms VI located on the **Functions»Waveform»Waveform Operations** palette on the block diagram. This function adds the two waveforms to obtain a noisy sinewave.



f. Place the FFT Power Spectrum VI located on the **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. To create the averaging parameters cluster control,

right-click that input terminal and select **Create»Control** from the shortcut menu.



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



- h. Place the Unbundle By Name function located on the **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»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 becomes True.



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

4. Save this VI as `Generate & Analyze Data.vi` into the `c:\exercises\LV Basics 2` directory.
5. Observe how the subVIs you used in this block diagram use error handling. Double-click the Sine Waveform VI and open its block diagram. Notice that the first thing it does is to check 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.
6. Run the VI. You can adjust the front panel controls to see the time and frequency waveforms change. You can force an error by entering the wrong values into these controls. For example, a sampling frequency, F_s , too low or too high results in an error.

By turning the averaging mode on, you can extract the sine wave peak from the noise regardless of their specified amplitudes. Notice what affect the different averaging techniques have on the signal.

7. Stop and close this VI when you are finished.

End of Exercise 1-1

D. LabVIEW Programming Architectures

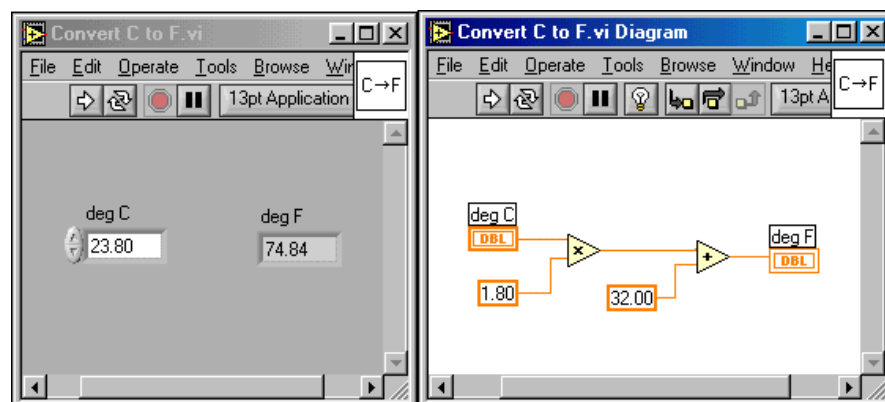
You can develop better programs in LabVIEW and in other programming languages if you follow consistent programming techniques and architectures. Structured programs are easier to maintain and understand. Now that you have created several VIs in LabVIEW through either the LabVIEW Basics I course or from similar programming experience, this concept of structured programming is described in more detail.

One of the best ways to create a program architecture that is easy to understand is to follow modular programming techniques and make subVIs for reusable or similarly-grouped operations. For example, refer to the VI you built in Exercise 1-1. The subVIs you used make the VI very easy to follow and understand while each piece can be reused in other VIs. Combined with documentation on the block diagram and in the **File»VI Properties»Documentation** option, a modular VI is easy to understand and modify in the future.

You can structure VIs differently depending on what functionality you want them to have. This section describes some of the common types of VI architectures, along with their advantages/disadvantages—simple, general, parallel loops, multiple cases, and state machines.

Simple VI Architecture

When doing 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 and can be initiated when the user clicks the run arrow. In addition to being commonly used for simple applications, this architecture is used for functional components within larger applications. You can convert these simple VIs into subVIs that are used as building blocks for larger applications.



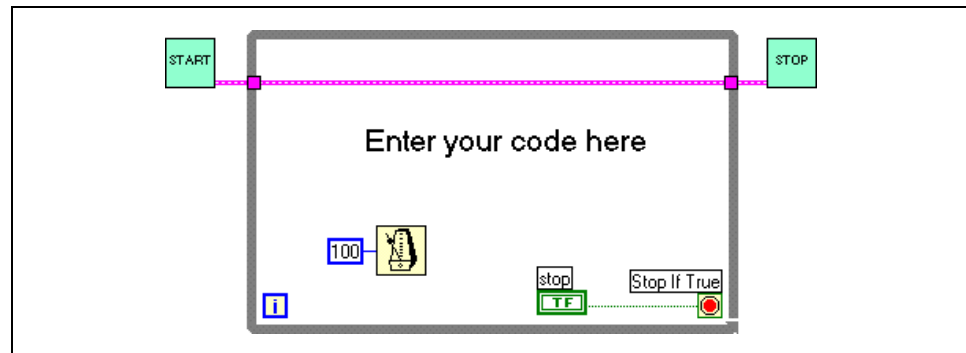
The previous front panel and block diagram example is the Convert C to F VI built in the LabVIEW Basics I course. This VI performs the single task of converting a value in degrees Celsius to degrees Fahrenheit. You can use this simple VI in other applications that need this conversion function without needing to remember the equation.

General VI Architecture

In designing an application, you generally have up to three main phases:

Startup	This section is used to initialize hardware, read configuration information from files, or prompt the user for data file locations.
Main Application	This section generally 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 section usually takes care of closing files, writing configuration information to disk, or resetting I/O to its default state.

The following block diagram shows this general architecture. For simple applications, the main application loop can be fairly straightforward. When you have complicated user interfaces or multiple events, such as, user action, I/O triggers, and so on, this section can get more complicated. The next few illustrations show design strategies you can use to design larger applications.

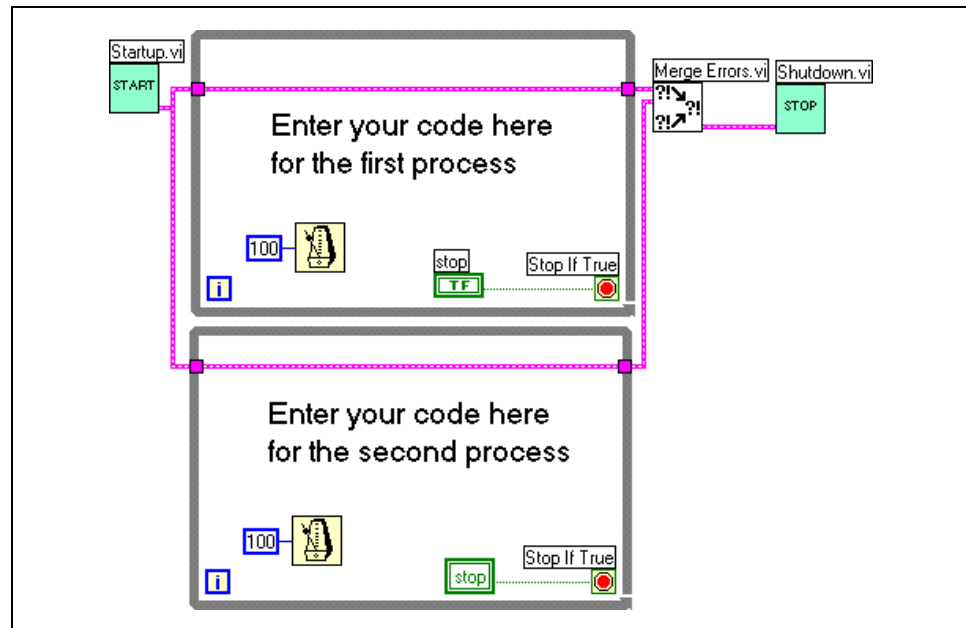


Notice in the previous block diagram that the error cluster wires control the execution order of the three sections. The While Loop cannot begin running until the Startup VI is finished running and returns the error cluster. Consequently, the Shutdown VI cannot run until the main program in the While Loop is finished and the error cluster is passed out of the loop. Another thing to notice in the previous block diagram is the wait function. A 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 so that it uses all of the computer's system resources. The wait function forces the loop to run asynchronously even if the wait period is specified as zero milliseconds. If the operations inside the main loop react to user inputs, then the wait period can be increased to a level acceptable for reaction times. A wait of 100–200 ms is usually good as most users will not detect that amount of delay between pushing a button on the front panel and the subsequent event executing.

Parallel Loop VI Architecture

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



This structure is straightforward and appropriate for some simple menu type VIs where a user is expected to select from one of several buttons that lead to different actions. This VI architecture also has an advantage over other techniques in that taking care of one event does not prevent you from responding to additional events. For example, if a user selects a button that causes a dialog box to appear, parallel loops can continue to respond to I/O events. Therefore, the main advantage of the Parallel Loops VI architecture is its ability to handle simultaneous multiple independent processes.

The main disadvantages of the parallel loop VI architecture lie in coordinating and communicating between different loops. The **Stop** button

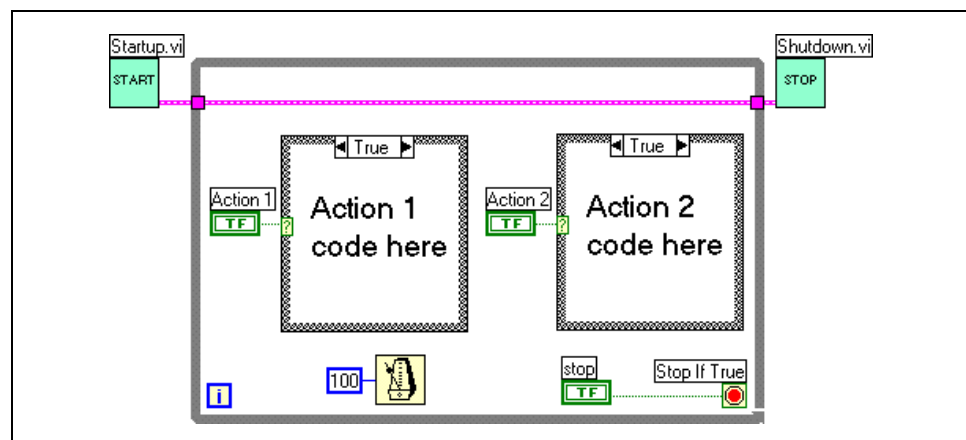
for the second loop in the previous block diagram is a local variable. You cannot use wires to pass data between loops, because that would prevent the loops from running in parallel. Instead, you have to use some global technique for passing information between processes. This can lead to race conditions where multiple tasks attempt to read and modify the same data simultaneously resulting in inconsistent results and are difficult to debug.



Note Refer to Lesson 3, *Data Management Techniques*, of this course for more information about global variables, local variables, and race conditions.

Multiple Case Structure VI Architecture

The following block diagram shows how to design a VI that can handle multiple events that can pass data back and forth. Instead of using multiple loops, you can use a single loop that contains separate case structures for each event handler. This VI architecture would also be used in the situation where you have several buttons on the front panel that each initiate different events. The following FALSE cases are empty.



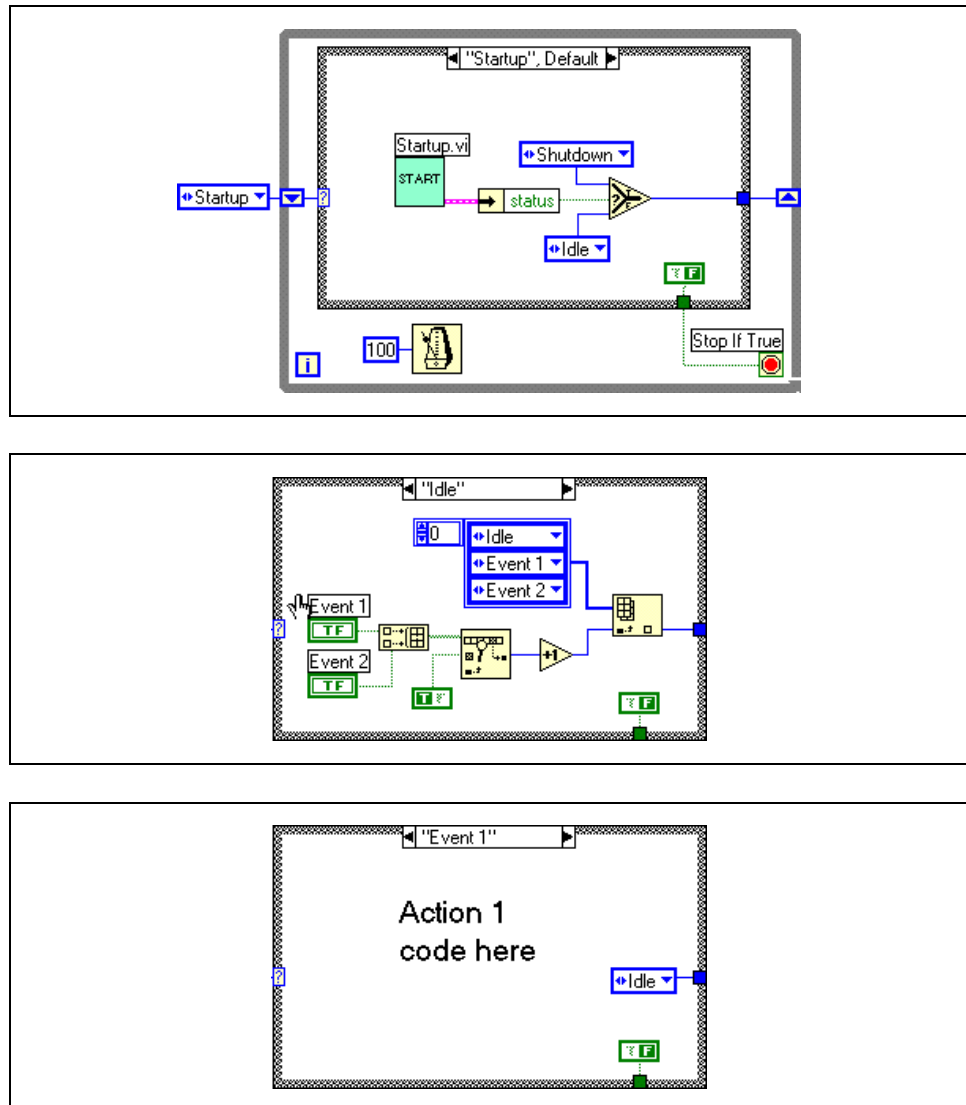
An advantage of this VI architecture is that you can use wires to pass data. This helps improve readability. This also reduces the need for using global data, and consequently makes it less likely that you will encounter race conditions. You can use shift registers on the loop border to remember values from one iteration to the next to pass data as well.

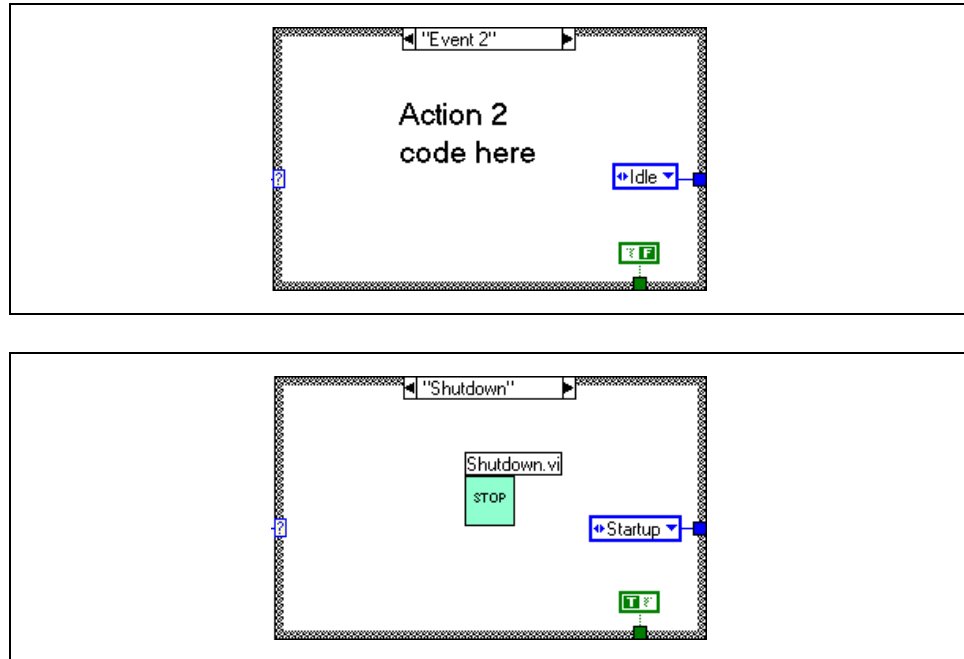
Several disadvantages exist to the multiple case structure VI architecture. First, you can end up with block diagrams that are very large and consequently are hard to read, edit, and debug. In addition, because all event handlers are in the same loop, each one is handled serially. Consequently, if an event takes a long time, your loop cannot handle other events. A related problem is that events are handled at the same rate because no event can repeat until all objects in the While Loop complete. In some applications, you might want to set the priority of user interface events to be fairly low compared to I/O events.

State Machine VI Architecture

You can make the block diagrams more compact by using a single case structure to handle all of your events. The State Machine VI architecture is a method for controlling the execution of VIs in a nonlinear fashion. This programming technique is very useful in VIs that are easily split into several simpler tasks, such as VIs that act as a user interface.

You create a state machine in LabVIEW with a While Loop, a Case structure, and a shift register. Each state of the state machine is a 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 appears in the following figure.





In this architecture, you design the list of possible events, or states, and then map that to each case. For the VI in the previous block diagram, the possible states are startup, idle, event 1, event 2, and shutdown. These states above are stored in an enumerated constant. Each state has its own case where you place the appropriate nodes. While a case is running, the next state is determined based on the current outcome. The next state to run is stored in the shift register. If an error occurs in any of the states, the shutdown case is called.

The advantage of the State Machine VI architecture is that the block diagram can become much smaller, making it easier to read and debug. One drawback of the Sequence structure is that it cannot skip or break out of a frame. The State Machine architecture solves that problem because each case determines what the next state will be as it runs.

A disadvantage to the State Machine VI architecture is that with the approach in the previous block diagram, you can lose events. If two events occur at the same time, this model handles only the first one, and the second one is lost. This can lead to errors that are difficult to debug because they can occur only occasionally. More complex versions of the State Machine VI architecture contain extra code that builds a queue of events, states, so that you do not miss any events.

More About Programming Architecture

As with other programming languages, many different methods and programming techniques are used when designing a VI in LabVIEW. The VI architectures shown in this section are some of the common methods to give you an idea of how to approach writing a VI.

The VI structures will get much more complicated as applications get larger and many different hardware types, user interface issues, and error checking methods are combined. However, you will see these same basic programming architectures used. Examine the larger examples and demos that ship with the LabVIEW application and write down which common VI architecture is used and why. Additional resources for making LabVIEW applications are described in the *LabVIEW Development Guidelines* manual.

Next you will build a VI that uses the Simple VI architecture to verify the name and password of a user.

Exercise 1-2 Verify Information VI

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

You will 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.



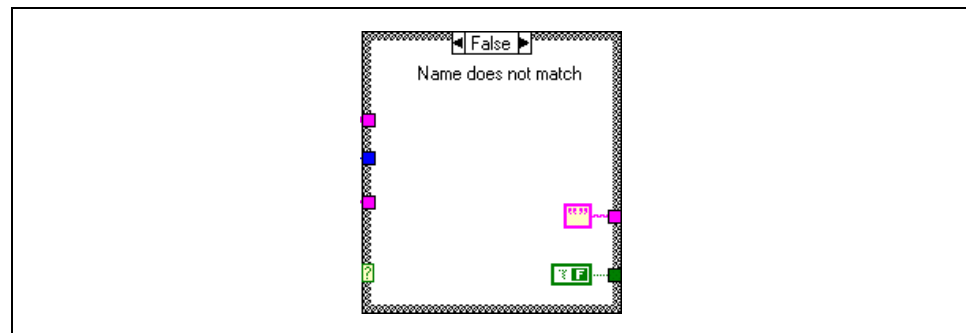
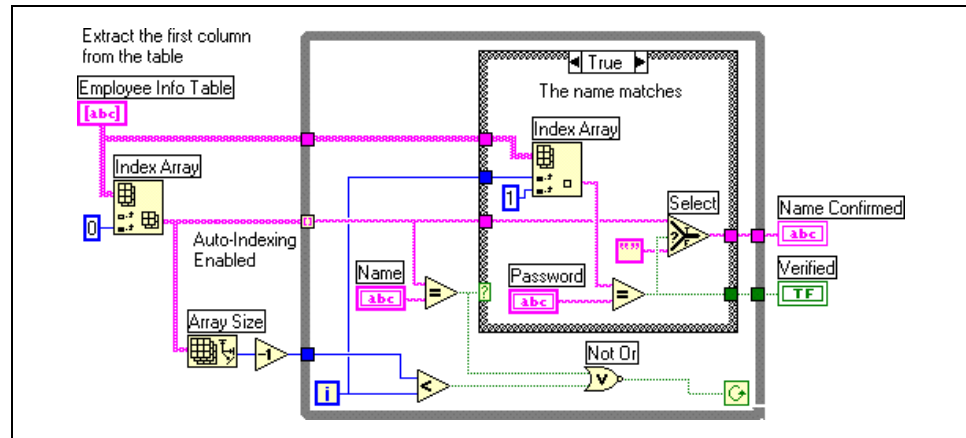
Note You will use this VI in Lesson 3.

Front Panel



1. Open a new VI and build the previous front panel. Modify the string controls as described in the labels by right-clicking the control. The Table control located on the **Controls»List & Table** palette is a two-dimensional array of strings where the first cell is at element 0,0.
2. Enter the information shown in the previous front panel 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.

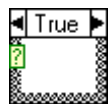
Block Diagram



3. Open and build the previous block diagram using the following components.



- a. Place a While Loop located on the **Functions»Structures** 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. Notice that indexing is enabled on the names array but not on the other two items entering the While Loop. Enable the indexing by right-clicking the tunnel and selecting **Enable Indexing** from the shortcut menu.



- b. Place a Case structure located on the **Functions»Structures** palette on the block diagram. If the **Name** input matches a listing in the first column of the table, the Passwords are also checked. If the **Name** does not match the current table entry, the loop continues to the next iteration.

To create the Boolean constant in the False case, wait until the True case is completely wired. Then select the False case and right-click the green outlined tunnel and select **Create»Constant** from the shortcut menu.



- c. Place the Index Array function located on the **Functions»Array** palette on the block diagram. This function is used to pull the names

array out of 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. You will use two of these functions, so make a copy of the Index Array function and place it into the True Case.



- d. Place the Array Size function located on the **Functions»Array** palette on the block diagram. This function returns the size of the names array.



- e. Place the Decrement function located on the **Functions»Numeric** palette on the block diagram. This function decreases the number of names in the array by one so that it can control the While Loop which starts indexing at 0.



- f. Place the Equal? function located on the **Functions»Comparison** palette on the block diagram. You will use two of these functions to check if the **Name** input matches a table entry and if the Password also matches the table entry.



- g. Place the Less Than? function located on the **Functions»Comparison** palette on the block diagram. This function controls the While Loop conditional. The loop continues to run while the current iteration number is less than the number of rows in the table.



- h. Place the Not Or function located on the **Functions»Boolean** palette on the block diagram. This function also controls the conditional of the While Loop so that the loop continues until a match is found or there are no more rows in the table.



- i. Place the Empty String constant located on the **Functions»String** palette on the block diagram. You will use two of these functions, one in the True case and one in the False case. If the names match but not the passwords or if neither match, then an empty string is returned in the **Name Confirmed** indicator.



- j. Place the Select function located on the **Functions»Comparison** palette on the block diagram. This function is used along with the password matching. If the password matches, the current name is sent to the **Name Confirmed** indicator; otherwise, the Empty String is sent.

4. Save this VI as `Verify Information.vi`.
5. Go to the front panel and make sure you have some names—the names do not have to be the same ones used in the course manual—and passwords in the table control. If you have not already done so, right-click the table and select **Data Operations»Make Current Values Default** so that the names and passwords are permanently stored in the table.

6. Enter values into the **Name** and **Password** controls and run the VI.

If the **Name** and **Password** match one of the rows in the table, the name is returned in the **Name Confirmed** indicator and the **Verified LED** is lit. Otherwise, an empty string is returned and the Verified LED is off. Be sure to try various combinations of names and passwords to make sure that this VI shows the correct behavior.

7. Create an icon for this VI because you will use this VI as a subVI in a later exercise. To create the icon, right-click the icon in the top right corner of either the front panel or the block diagram and select **Edit Icon** from the menu. Design an icon similar to the one shown here.



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



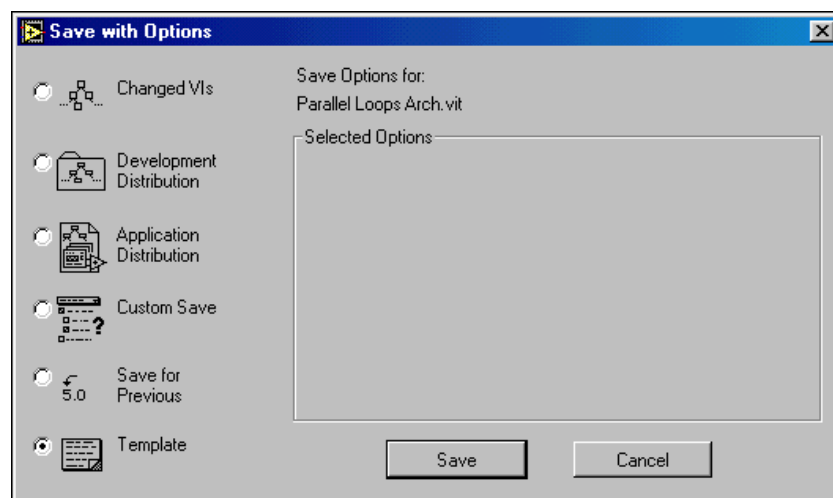
9. Save the VI under the same name.
10. Stop and close this VI when you are finished.

End of Exercise 1-2

E. VI Templates

In the last section, you learned about some of the common programming architectures for VIs. So that you do not have to start building each new VI from an empty front panel and block diagram, a number of templates are included with the LabVIEW package. These VI templates can be found in the `LabVIEW 6\Templates` directory and contain the file extension `.vit`. LabVIEW provides some basic templates, however, not all of the VI architectures described are available as templates.

You can save a VI you create as a template by using the **File»Save with Options** option as shown in the following dialog box.



This option allows you to save VI architectures and other programming structures you use often for your own templates. Now you will examine a couple of the template VIs.

Exercise 1-3 Timed While Loop with Stop VIT, State Machine VIT

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

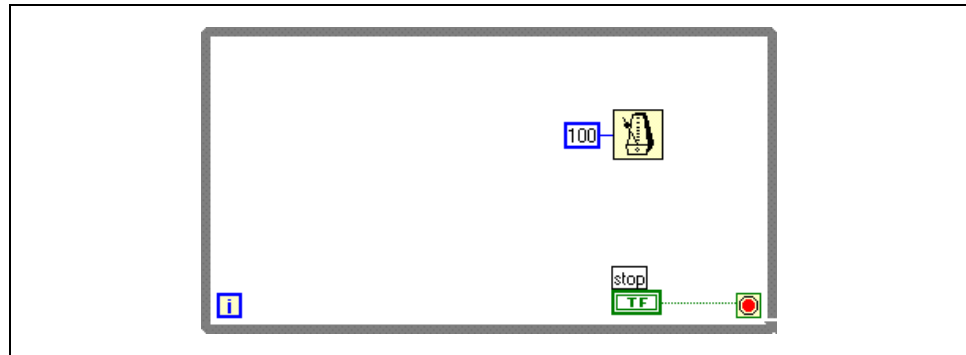
You will examine template VIs that show both the State Machine architecture and the General architecture.

Front Panel



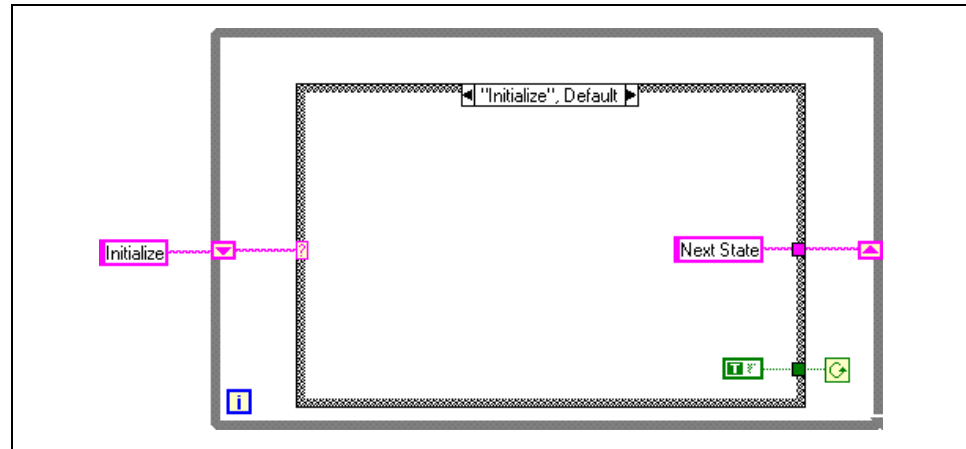
1. Open the `Timed While Loop with Stop.vit` from the LabVIEW 6\Templates directory. The front panel only contains the **Stop** button.

Block Diagram



2. Open and examine the block diagram.
3. You will recognize this VI as having the General VI architecture. It contains a While Loop that stops when you click the button on the front panel and a wait function in the loop ensures that this loop will not use all the system resources.
4. Return to the front panel and run the VI. It does nothing but continues to run until you click the **Stop** button.
5. Stop and close this VI when you are finished.
6. Open the `State Machine.vit` from the LabVIEW 6\Templates directory. The front panel is empty.

Block Diagram



7. Open and examine the block diagram.
8. This State Machine VI architecture is implemented in a slightly different manner as the one previously described. This VI uses a string constant to contain the states for the Case structure whereas the previous VI showed an enumerated type, similar to a ring control, to control the states. Because the Case structure accepts both numeric and string data, it does not matter how you specify the different states. The key to the State Machine architecture is that you have a While Loop with a Case structure inside where each case is a different state of the overall application. The next state is determined while the VI is running based upon what happens in the current state.
9. Close this VI when you are finished.

End of Exercise 1-3

Summary, Tips, and Tricks

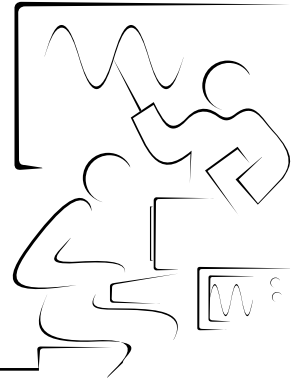
- In most cases, a top-down approach is used to plan the overall strategy for a project. Development and implementation of an application usually occurs from the bottom-up.
- When designing a LabVIEW application, it is important to determine the end-user's expectation, exactly what the application must 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. By working with small modules, you can debug an application by testing each module individually. This approach also makes it much easier to modify the application in the future.
- 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.
- The most common VI architectures are the Simple, the General, the Parallel Loops, the Multiple Case Structures, and the State Machine. Each of these architectures has advantages and disadvantages depending upon what you want your application to do.
- The State Machine VI architecture is very useful for user interface VIs and it generates clean, simple code.
- Several template VIs ship with the LabVIEW package so you do not need to start VI development from empty front panels and block diagrams each time.
- Refer to the *LabVIEW Development Guidelines* manual for more information about designing and building VIs.

Notes

Notes

Lesson 2

Designing Front Panels



This lesson introduces how you can design and build the user interface to your VIs. First, an overview and list of things to consider while building the front panel. The next few sections describe some common methods and tools for customizing panels. The last few sections describe additional items and panel objects used for developing user interfaces.

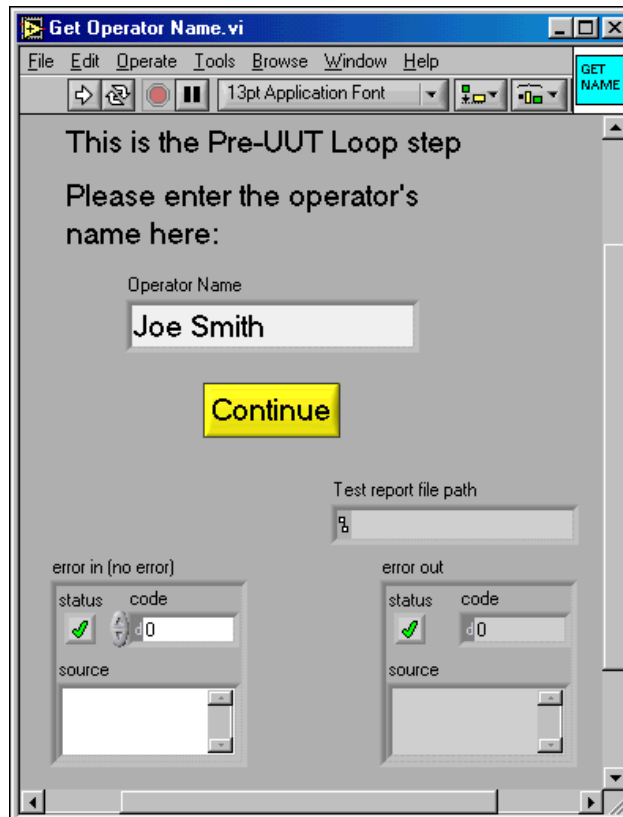
You Will Learn:

- A. About basic user interface issues.
- B. How to use Boolean clusters as menus.
- C. About Property Nodes.
- D. About Graph and Chart Properties.
- E. How to use Control References.
- F. About LabVIEW Run-Time Menus.
- G. About Intensity Plots.

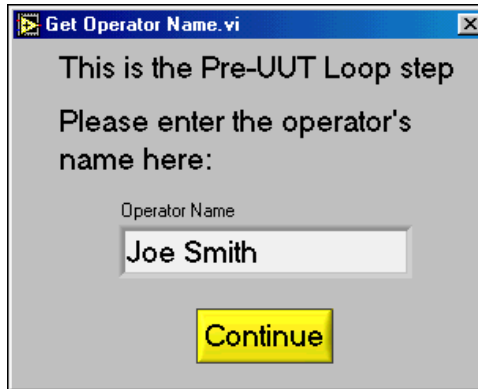
A. Basic User Interface Issues

When you develop applications that other people will use, you need to follow some basic rules regarding the user interface. 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 not receive important information from the data. This section describes some topics to consider when you are building front panels.

One rule to follow when building a user interface to a VI is to only show items in the front panel that the user needs to see at that time. The example following 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.



The next example shows the same subVI after the front panel has been resized and the menu bar, scrollbars, and toolbar have been removed with the **VI Properties»Window Appearance»Dialog** option.



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. Here are some color matching tips:

- Start with a gray scheme—select one or two shades of gray and highlight colors that have good contrast with 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 spacing and alignment to group objects instead of matching colors.
- Good places to learn about color are stand-alone instrument panels, maps, magazines, and nature.

Also, keep in mind that approximately twenty percent of engineers are color-blind to some degree. They rely on contrast more than color when differentiating items.

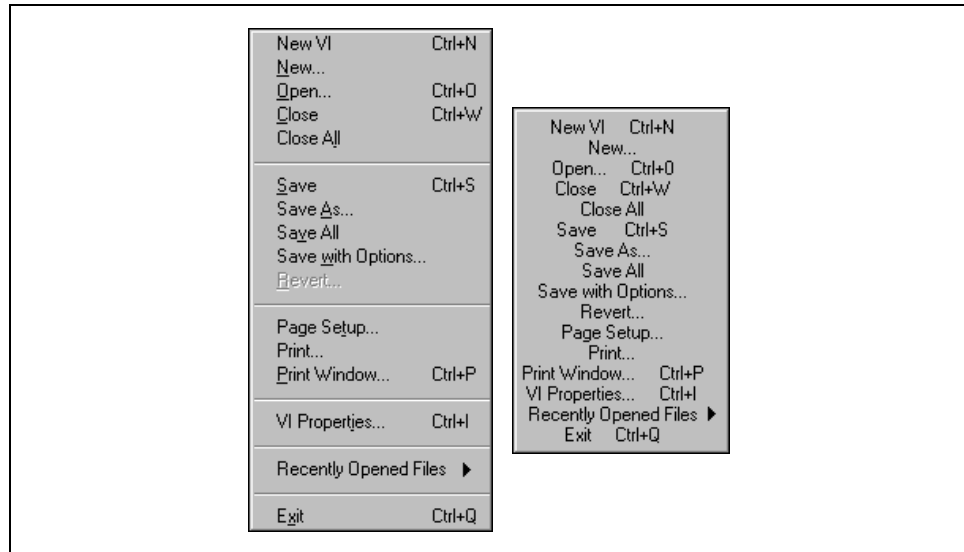
Spacing and Alignment

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 might see items right to left, almost all follow top to bottom.

Centered items are better than random, but much less orderly than either left or right. 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.

Menus are left-justified and related shortcuts right-justified as shown in the bottom left example of the LabVIEW **File** menu. It is more difficult to locate

items in the center-justified menu as shown in the bottom right example. Notice how the simple dividing lines between menu sections help you find the items quickly and strengthen the relationship between the items in the section.



Text and Fonts

Text is easier to read and information makes more sense when displayed in an orderly way. To use this to your advantage when building front panels, try grouping related controls together with white space or with lines. These methods work best when they are subtle and do not distract from the information.

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 quite different. If not, it will look like a mistake. Similarly, if you use two different fonts, make sure they are distinct.

Operator stations can catch lots of glare or users might need to read them from a greater distance than normal computers or with a touch screen. Therefore, you should use larger fonts and more contrast.

User Interface Tips and Tools

It is not necessary or wise to spend time making every front panel look polished—concentrate on the ones that users see the most. 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.

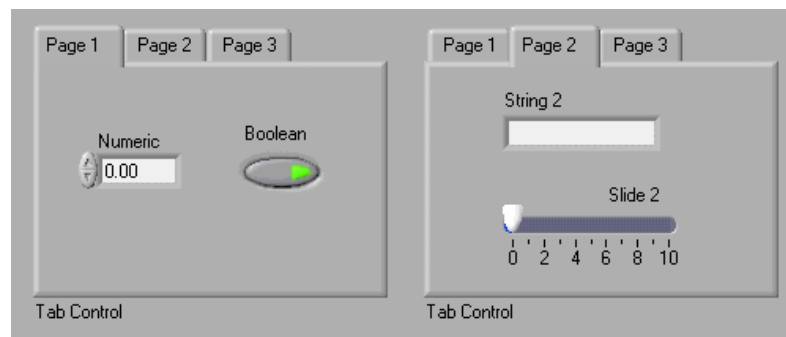
Dialog Controls

A common user interface technique is to have dialog boxes appear at appropriate times to interact with the user. You can make a VI behave like a dialog box by checking that option in the **File»VI Properties»Window Appearance** dialog box. The **Controls»Dialog Controls** palette contains the same kinds of objects available in system dialog boxes.

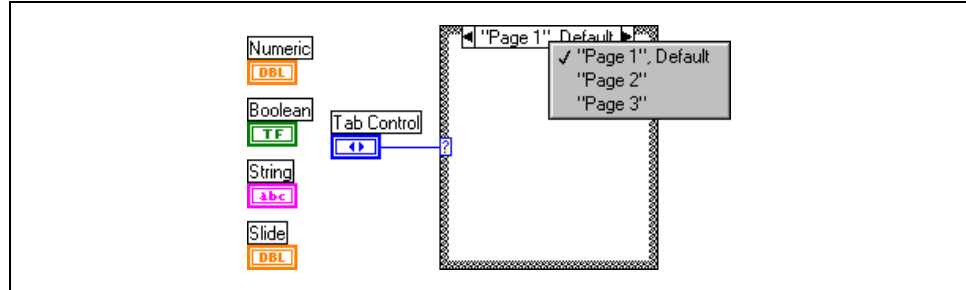
You should use these dialog controls to build system-like dialogs. Dialog controls change appearance according to the operating system and Window Appearance settings and they typically ignore all color clicks except transparent. If you are integrating a graph or non-dialog controls into the front panel, 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 them, but move to smaller or more efficient controls, such as rings, or tab controls where this makes sense. Tab Controls are found in the **Controls»Dialog Controls** palette or the **Controls»Array & Cluster** palette and offer a convenient way to group front panel objects together.



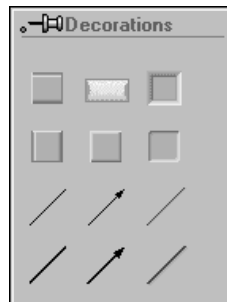
The way you use a Tab Control is to place it on the front panel as shown previously. You add tabs by right-clicking an existing tab and selecting **Add Page After**, **Add Page Before**, or **Duplicate Page** from the menu. You relabel the tabs with the Labeling tool, and you place other front panel objects into the appropriate pages. The terminals for these objects are available in the block diagram as are terminals for any other front panel object.



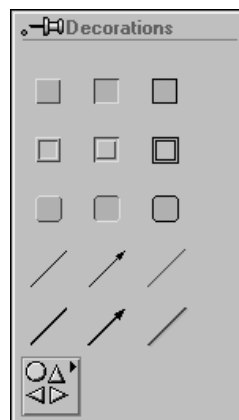
The Tab Control wires directly to a Case structure as shown previously and the page name shows up as the name of the case. You can then wire the data to and from the other terminals as needed.

Decorations

One of the easiest and often overlooked methods of grouping or separating objects is to use an item from the **Controls»Decorations** palette. You can select to use various types of boxes, arrows, and lines from this palette as follows:



You can find more decorations in the **Controls»Classic Controls** palette. The following palette shows the various boxes, borders, lines, arrows, circles, and triangles:



Menus

Menus are a good way to present most of a front panel's functionality in an orderly way and in a relatively small space. This leaves room on the front panel for actions that are needed in an emergency, items for beginners, items needed for productivity, and items that do not fit well into menus. Also be sure to add menu shortcut keys for the frequently accessed items. Later in this lesson you will learn how to create your own menus in a VI.

Automatic Resizing of Front Panel Objects

As mentioned previously, you can use the **VI Properties»Window Appearance** options to modify the appearance of the front panel when a VI is running. With the **VI Properties»Window Size** options, you can set the minimum size of a window, keep the window proportion with screen changes, and set front panel objects to resize in two different modes. Most professional applications do not enlarge each and every control when the window changes size, but pick a table, graph, or list to enlarge with the window, leaving other objects near the window's edge. To resize one front panel object with the front panel, select that object, and select **Edit»Scale Object With Panel**.

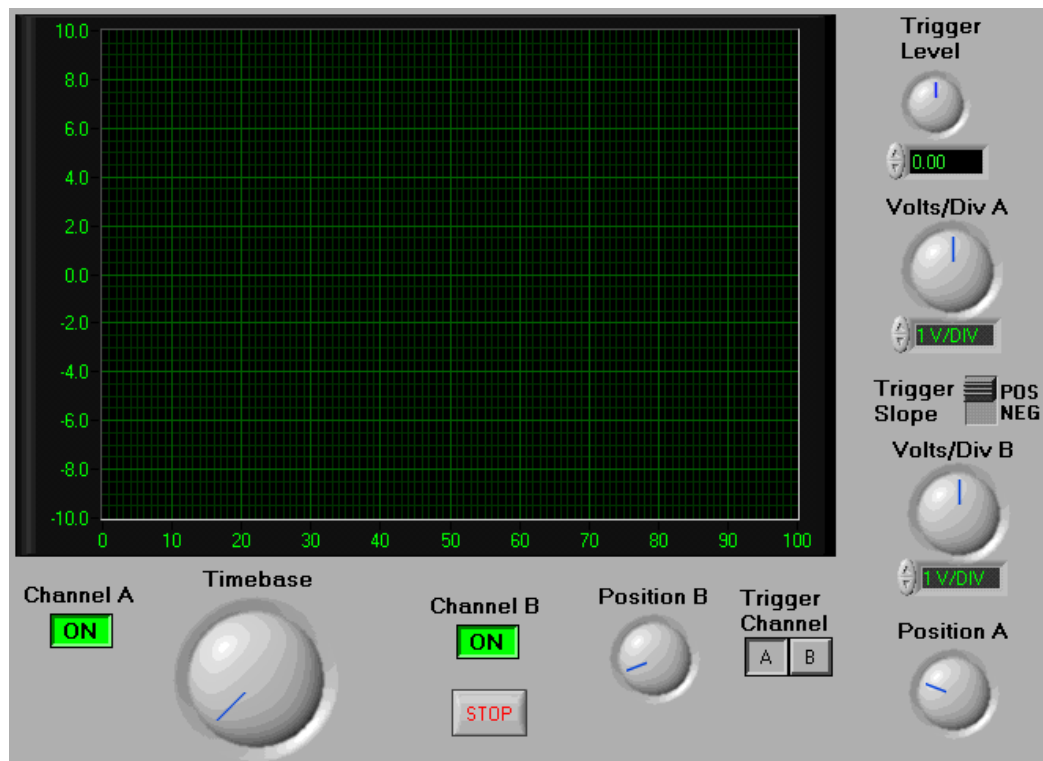
The following exercise uses some of the techniques described to create a VI with a user interfaces that is easy to use.

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.

You will resize, reorganize, and rearrange the objects on the front panel to make the user interface easier to use. You will also setup the graph to resize along with the front panel.

Front Panel



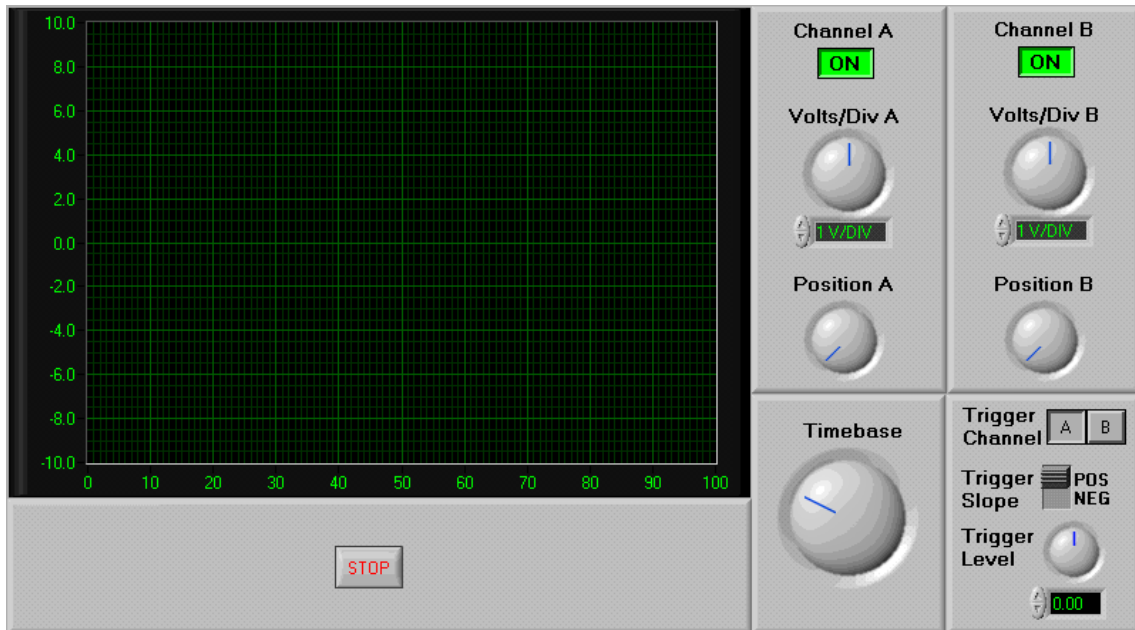
1. Open the Scope Panel VI from the `c:\exercises\LV basics 2` directory. The front panel is shown previously.
2. Move the controls around as to logically group the controls that share similarities (not all can relate to a group). For example the **Channel A** ON/OFF button, Channel A Position knob, and Channel A Volts/Div knob all operate on channel A and it makes sense to have them close to one another. Two other example groups would be the three Channel B groups and the three trigger controls.



Tip Remember to use the Align Objects and Distribute Objects features in the tool bar.

3. After making groups out of the controls, use the Raised Box decoration on the **Controls»Decorations** palette to make visible separations

between the groups. And resize your window so the front panel fits inside the window, as shown in the following example.



Tip You need to use the **Reorder** button in the tool bar on the decorations to Move to Back, so that the controls are visible and on top of the raised boxes.

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 **OK** 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 under the same name.
7. Resize the window. You notice that the graph does resize with the window and the controls maintain a proportional distance to the graph and each other but do not grow or shrink. However, you notice that the decorations do not resize. In order to do this, you must use Boolean buttons that look the same in each state and resize them programmatically. This requires the use of Property Nodes and is covered later in this lesson.
8. Close the VI when you are finished.

End of Exercise 2-1

Exercise 2-2 Acquire Data.vi

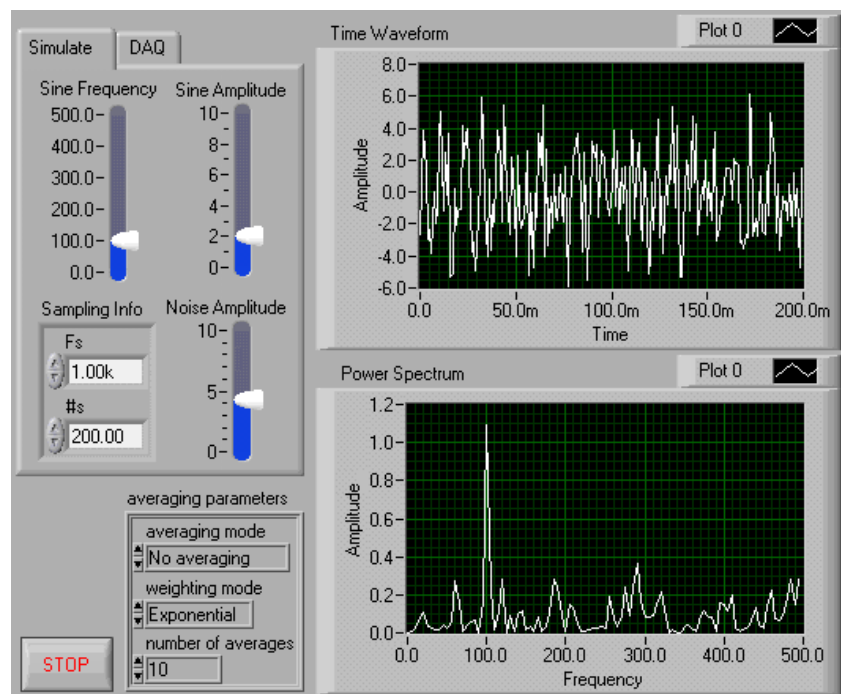
Objective: To modify a VI to use the Tab Control along with proper user-interface design techniques.

You will modify the Generate & Analyze VI so that it uses the tab control. One page of the tab will be for the generation of data as before and a second page will be for acquiring the data from a DAQ device.



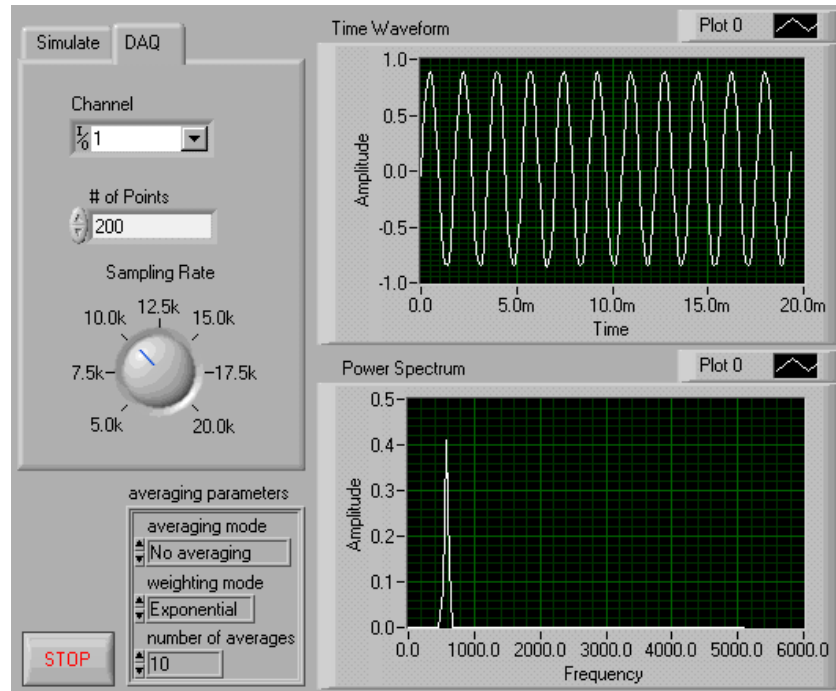
Note You will use this VI in the project in Lesson 5.

Front Panel



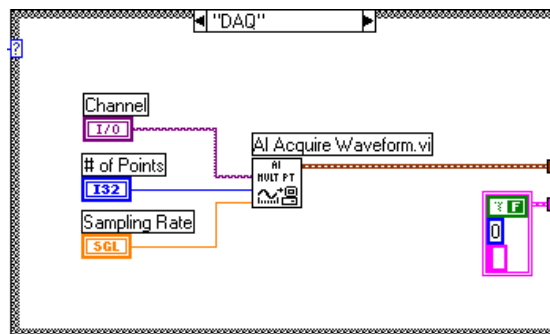
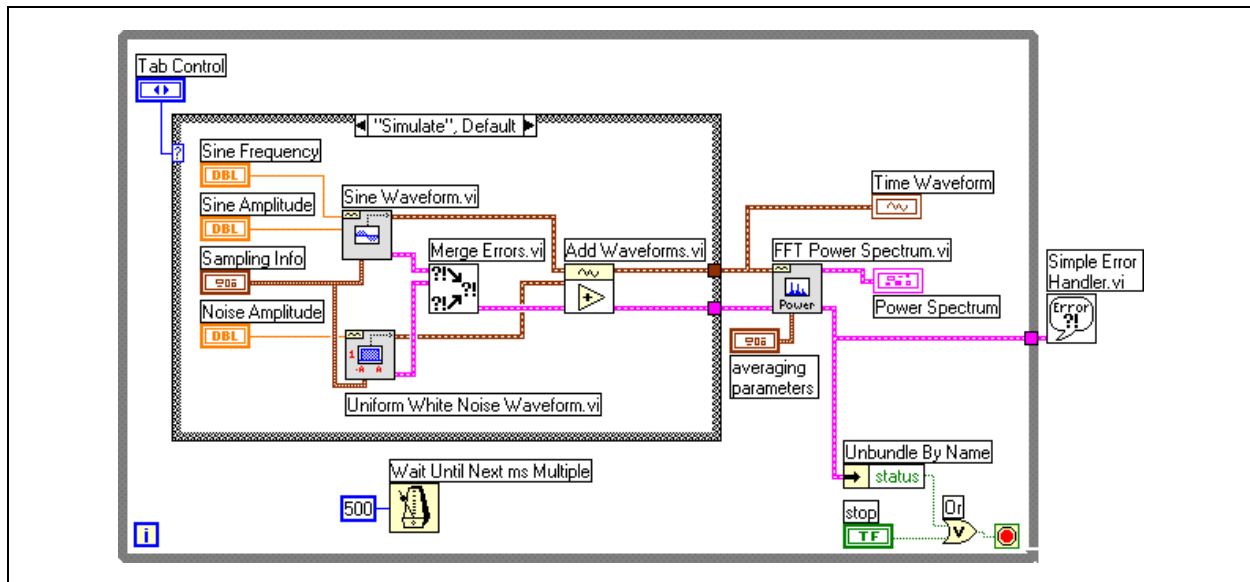
1. Open the Generate & Analyze VI you built in Exercise 1-1. Resize the front panel to make room on the left of the other front panel objects as you will be adding the tab control.
2. Place a Tab Control located on the **Controls»Array & Cluster** palette on the front panel as shown previously. Select the Sine Frequency, Sine Amplitude, Sampling Info, and Noise Amplitude controls and place them into the first page of the Tab Control.

- Name the two pages of the tab control Simulate and DAQ respectively. Click the DAQ page and add the objects as follows:

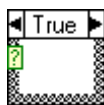


- Save this VI as `Acquire Data.vi`.
- Right-click the DAQ Channel Name control and make sure the **Allow Undefined Names** option is selected.
- To change the scale on the knob to read `20.0k` instead of `20,000`, right-click the knob and select **Format & Precision**. Select **Engineering Notation** and the `k` is added to represent one thousand.

Block Diagram



7. Open and modify the block diagram as shown previously using the following components:



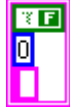
- a. Place a Case structure located on the **Functions»Structures** palette on the block diagram. When used with the tab control—one case handles the Simulate page and the other case is for the DAQ page.



- b. Place the AI Acquire Waveform VI located on the **Functions»Data Acquisition»Analog Input** palette on the block diagram. This VI acquires data from an analog input channel on the DAQ device.



Note If you do not have a DAQ device or a DAQ Signal Accessory, use the Demo Acquire Waveform VI located on the **Functions»User Libraries»Basics 2 Course** palette on the block diagram in place of the AI Acquire Waveform VI. The Demo Acquire Waveform VI simulates acquiring data from an analog input channel at a specified sampling rate and returning the specified number of samples.

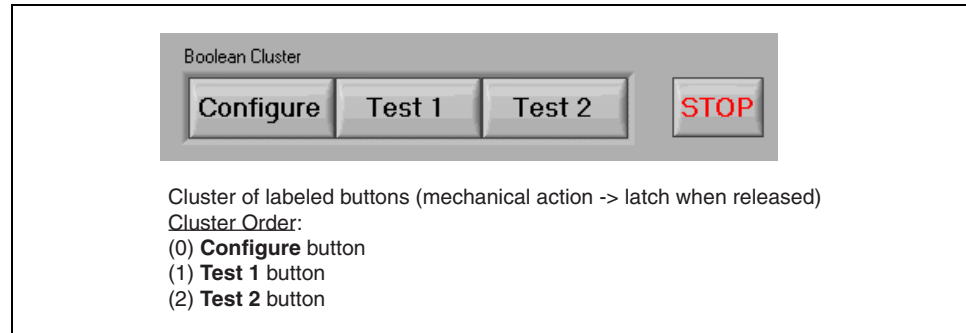


- c. Create a Cluster constant by right-clicking the Case structure tunnel and selecting **Create»Constant** from the shortcut menu. This constant passes the default values for the error cluster out of the DAQ case.
8. Save the VI.
9. Observe how you have added quite a bit of new functionality to this VI without adding a lot of extra code. Using the Tab Control is a very efficient way to add new front panel objects to the user interface and also add block diagram functionality without having to enlarge the front panel and block diagram windows.
10. Run the VI. You can adjust the front panel controls to see the time and frequency waveforms change. Click between the Simulated and DAQ pages in the Tab Control.
11. Stop and close this VI when you are finished.

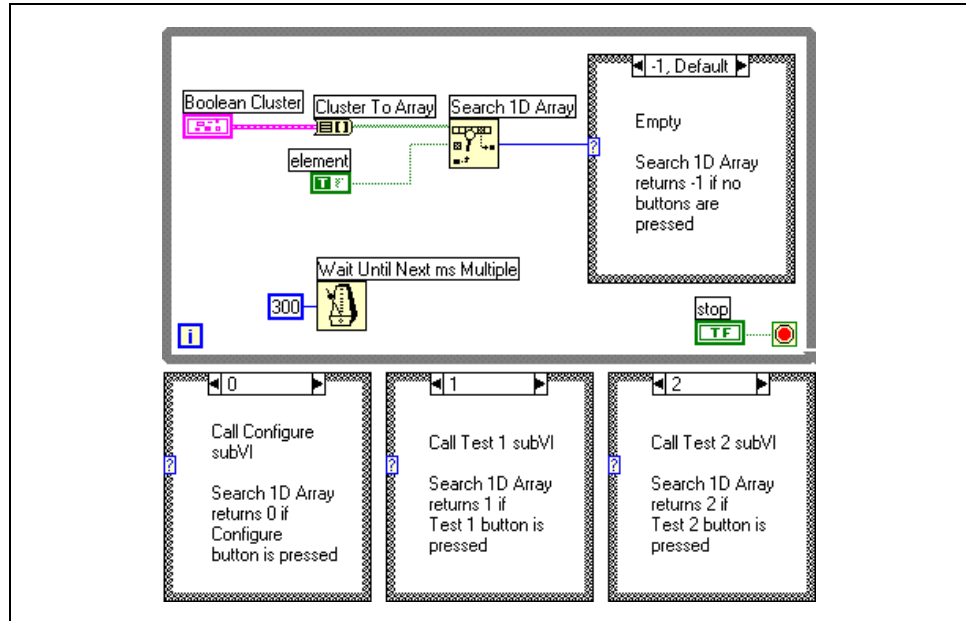
End of Exercise 2-2

B. Using Boolean Clusters as Menus

You can use latched Boolean buttons in a cluster to build a menu for an application. For example, consider an application where an operator configures a system and runs either of two tests. A possible menu VI for this application is as follows.



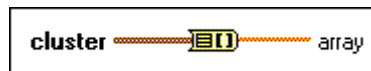
The Cluster to Array function converts the Boolean cluster to a Boolean array with three elements. That is, each button in the cluster represents an element in the array. The Search 1D Array function on the **Functions» Array** palette searches the 1D array of Boolean values created by the Cluster to Array function for a value of TRUE. A TRUE value for any element in the array indicates that you clicked on a button in the cluster. The Search 1D Array function returns the index of the first TRUE value it finds in the array. If you did not click a button, Search 1D Array returns an index value of -1. If no buttons are pressed, Case -1 is executed, which does nothing. Clicking the **Configure** button executes Case 0, which could, for example, call the Configure subVI. Clicking the **Test 1** button executes Case 1, which could call the Test 1 subVI, and clicking the **Test 2** button executes Case 2. The While Loop repeatedly checks the state of the Boolean cluster control until you click the **Stop** button. The VI block diagram is as follows.



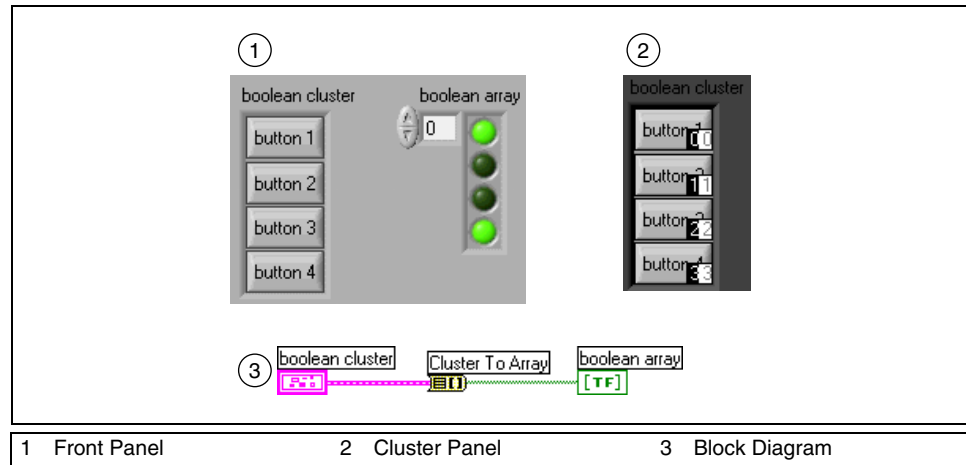
Cluster Conversion

You can convert a cluster to an array if all cluster components have the same data type (for example, all are Boolean data types or all are numeric). With this conversion, you can use array functions to process components within the cluster.

The Cluster to Array function **Functions»Cluster** and **Functions»Array** palettes converts a cluster of identically typed components to a 1D array of the same data type.



The following example 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) to the second element (index 1), and so on.



The Array to Cluster function (**Functions»Cluster** and **Functions»Array** palettes) converts a 1D array to a cluster in which each component in the cluster is the same type as the array element.

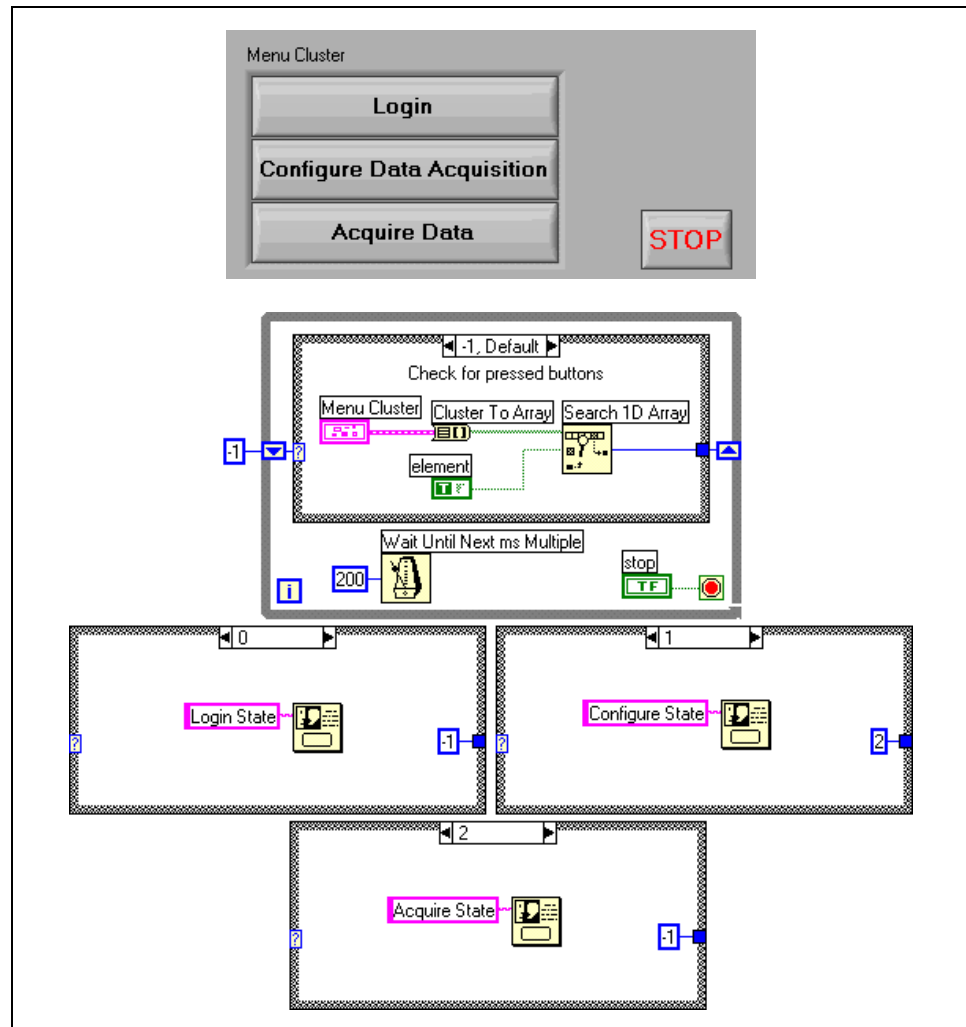


Note You must right-click the function icon to set the number of components in the cluster. The default number of components in the output cluster is nine.

You can combine the concept of a state machine with a Boolean menu cluster to provide a powerful system for menus. For example, perhaps you need to provide the following application, which can be divided into a series of states.

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 (0)
1	Configure	Configure acquisition	Acquire (2)
2	Acquire	Acquire data	No Event (0)

The following is an example of a state machine for this application.



The front panel consists of a Boolean button cluster, with each button triggering 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 will execute.

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 will be checked again, but in state 1 (Configure) the subsequent state is state 2 (Acquire).

Exercise 2-3 Menu.vi

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

A set of dependencies exist between the different operations of the application to be built in this course. Under most circumstances, after performing a certain action, the application should return to a “No Event” state, in which the application should monitor a menu to see which button should be 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.

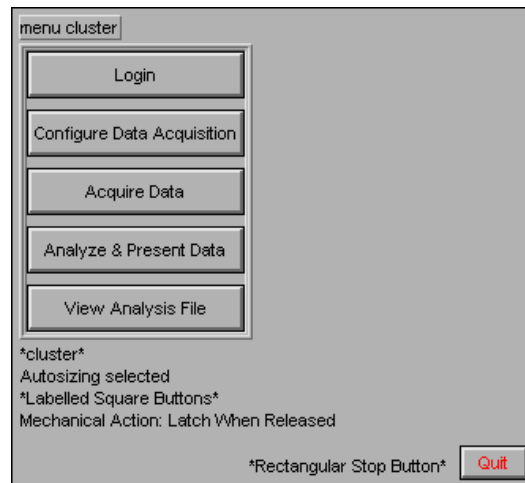
State Value	State Name	Description	Next State
-1, Default	No Event	Monitor Boolean menu to determine the next state	Depends on the Boolean button pressed. If no button is pressed, next state is No Event.
0	Login	Log in user	No Event
1	Acquire	Acquire data	No Event
2	Analyze	Analyze data, possibly save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

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



Note You will use this VI in Lesson 5.

Front Panel

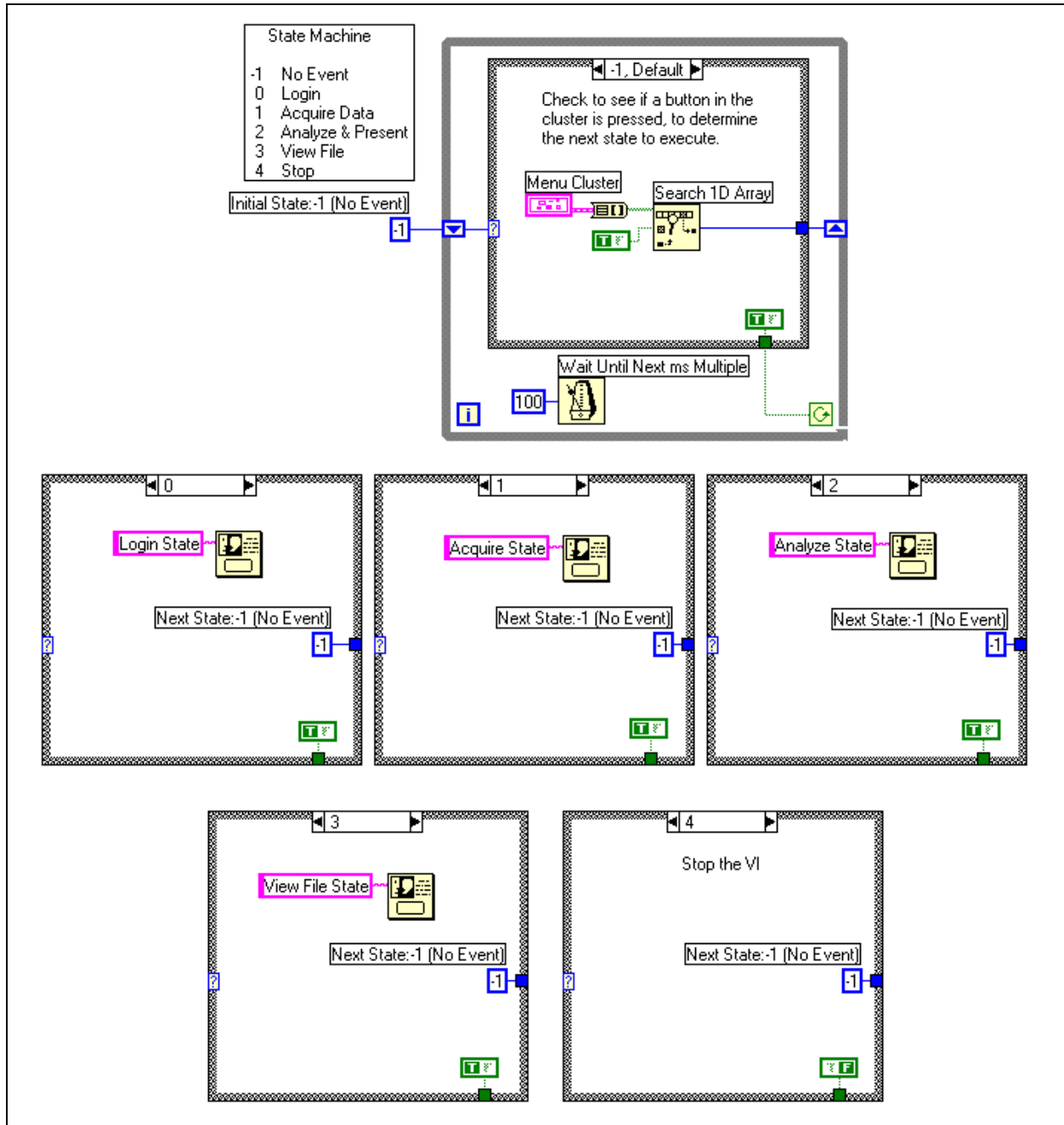


1. Open a new VI.
2. Build the front panel according to the previous example. Each button in the cluster will trigger an appropriate state when it is pressed. When building the front panel, make sure that the **Login** button is at cluster order 0. Acquire Data is cluster order 1, Analyze & Present Data is cluster order 2, View Analysis File is cluster order 3, and Stop is cluster order 4. The cluster order of the menu cluster will determine the numeric state which will be executed.

Hints:

- Create the button with the largest label first.
- Set the mechanical action to **Latch When Released**.
- Use **Copy** and **Paste** to create the other buttons.
- Use **Align** and **Distribute** to arrange the buttons.

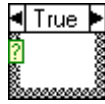
Block Diagram



1. Build the block diagram as shown previously.



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



- b. Place a Case structure located on the **Functions»Structures** palette on the block diagram. This structure makes the states for the state machine. Add cases by right-clicking the border of the Case structure. Be sure to define the -1 case as being the default.



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



- d. Place the Cluster To Array function located on the **Functions»Cluster** or **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.



- e. Place the Search 1D Array function located on the **Functions»Array** palette on the block diagram. In this exercise, this function searches the Boolean array that **Cluster to Array** returns for a TRUE value. A TRUE value for any element indicates that you clicked on the corresponding button. The function returns a value of -1 if you did not click a button.



- f. Place the One Button Dialog function located on the **Functions»Time & Dialog** palette on the block diagram. You will use four of these functions to indicate which state has been selected and loaded into the shift register.

2. Save the VI as `Menu.vi`.
3. Run the VI. When you click the **Login, Acquire Data, Analyze & Present Data**, or **Data File View** buttons, a dialog box appears to indicate that you are in the associated state.
4. 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 a button is pressed, however, 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 top of the exercise.

In later exercises, you will substitute VIs that you create for the One Button Dialog functions to build the application.

5. Click the **Stop** button on the VI's front panel to halt execution.
6. Close the VI when you are finished.

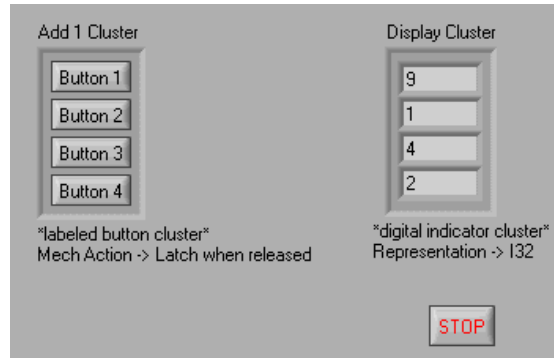
End of Exercise 2-3

Exercise 2-4 Cluster Conversion Example VI (Optional)

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

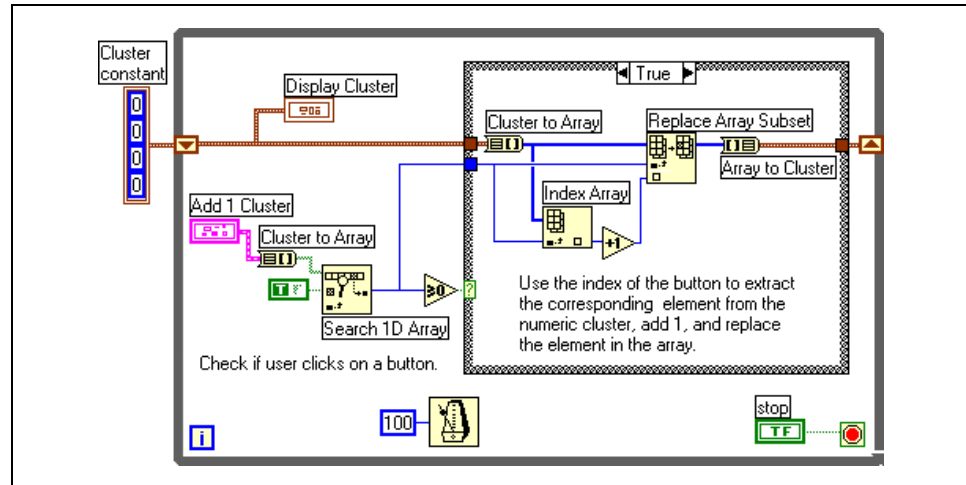
You will examine a VI that uses clusters to process data. The VI features a cluster containing four labeled buttons. The VI keeps track of the number of times you click each button.

Front Panel



1. Open the Cluster Conversion Example VI in the `c:\exercises\LV Basics 2` directory.

Block Diagram



False case is empty except for passing the cluster from the left shift register to the right shift register.

1. Open and examine the block diagram.
2. Run the VI. Click a button. The corresponding digital indicator should increment each time you click a button.
3. Close the VI. Do not save any changes.

End of Exercise 2-4

C. 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 to cause 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 threshold, you might want to show a red trace instead of a green one. Resizing front panel objects, hiding parts of the front panel, and adding cursors to graphs also can be done programmatically using Property Nodes.

Property Nodes in LabVIEW are very powerful and have many uses. This course only covers one use for Property Nodes to change the appearance and functional characteristics of front panel objects programmatically. You can set properties such as display colors, visibility, position, size, blinking, menu strings, graph or chart scales, graph cursors, and many more. Refer to the **Help»Contents and Index** in LabVIEW for more information about the Property Node.

Creating Property Nodes

You create Property Nodes by selecting the **Create»Property Node** option from the shortcut menu of a front panel object or from its terminal on the block diagram. Selecting **Create»Property Node** creates a new node on the block diagram near the terminal for the 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 create any number of Property Nodes for a front panel object.

Because there are many different properties for front panel objects, we will cover only some of the common properties. Select **Help»Contents & Index** to find information about a particular property.

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 read the current state of that property.

For example, if you create a Property Node for a digital control, it appears on the block diagram with the Visible property showing in its terminal. A small arrow appears on the right side of that terminal, indicating that you can read a value from the Visible property for the digital control. You can change it to a write property by right-clicking the node and selecting **Change To Write** from the shortcut menu. Wiring a Boolean FALSE to that terminal will cause the digital control to vanish from the front panel when the

Property Node receives the data. Wiring a TRUE causes the control to reappear.



You can read the current state of a property by right-clicking the Property Node terminal and selecting **Change To Read**. When the Property Node is called, it will output a Boolean TRUE if the control is visible or a Boolean FALSE if it is invisible.



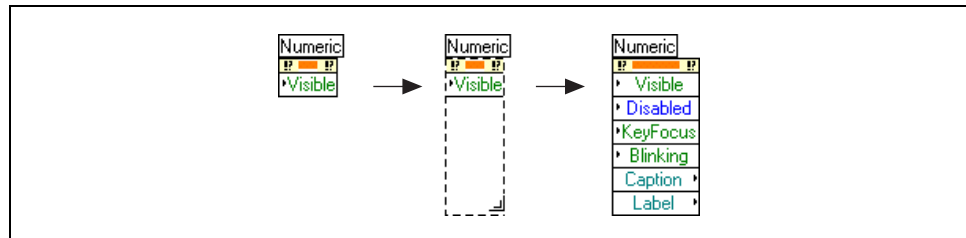
Operating tool

From the terminal on the Property Node, you can select properties by clicking the node with the Operating tool, shown at left, and selecting the desired property from the shortcut menu.

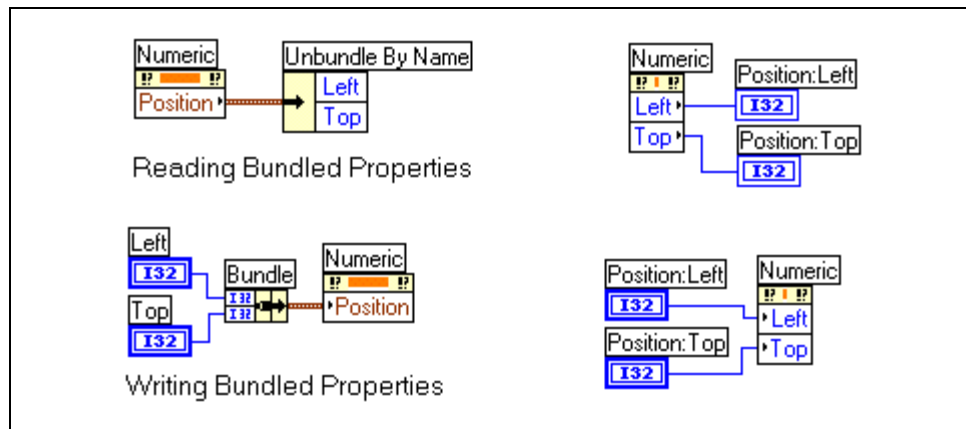


Positioning tool

You can read or set more than one property with the same node by enlarging the Property Node. Using the Positioning tool, shown at left, click the lower corner of the Property Node and drag the corner down to enlarge the node. As you enlarge the node, you add more terminals. You can then associate each Property Node terminal with a property from its shortcut menu.



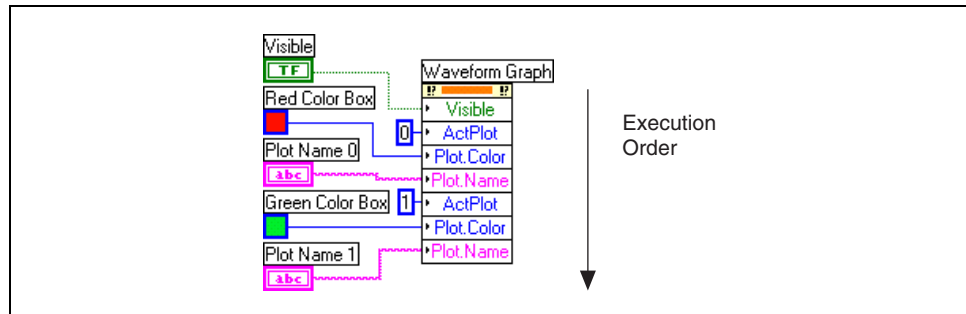
Some properties are clusters. These clusters contain several properties that you can unbundle using the Unbundle function located on the **Functions»Cluster** palette. Writing to these properties requires the Bundle function, as follows:



To access bundled properties, select the **All Elements** option from the property's shortcut menu. For example, you can access all the elements in the Position property by selecting **Position»All Elements** or you can access a single element from that bundle.

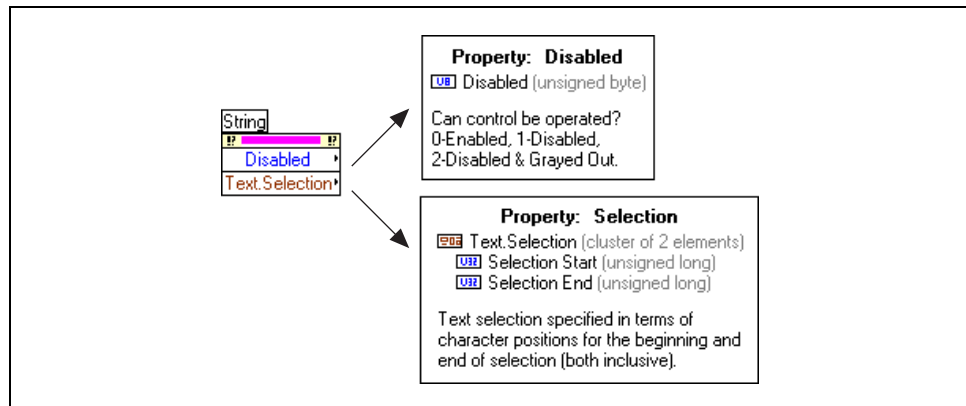
Property Node Execution Order

Property Node terminals execute from the top down. For example, in the waveform graph Property Node as follows, the graph is first made visible. Then Plot 0 is set as the active plot, its color set to red, and the name of the plot in the legend is updated. Finally, Plot 1 is set as the active plot, its color set to green, and the name of the plot in the legend is changed.



Using the Context Help Window

Use the LabVIEW Context Help window and the **Help»Contents and Index** to find descriptions, data types, and acceptable values for Property Nodes. With the Context Help window active, pass the cursor over terminals in the Property Node to display information. The following example shows help for both a single property and a cluster of several properties.



Common Properties

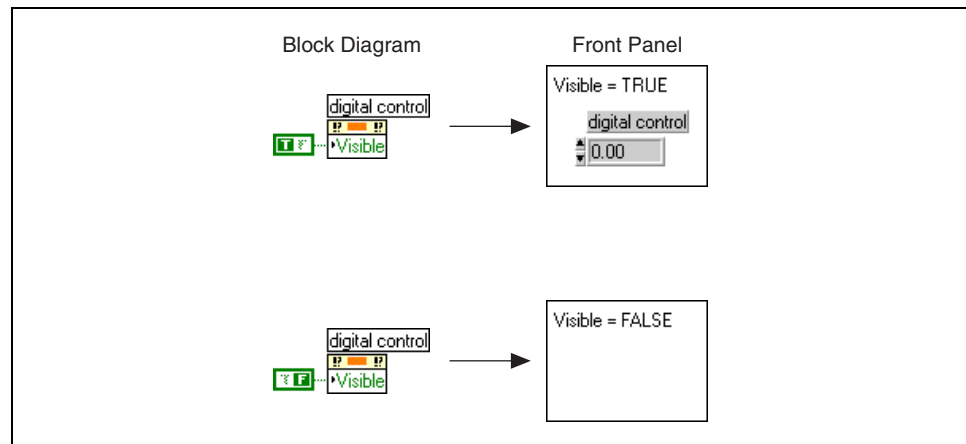
There are many properties available for the various front panel objects in LabVIEW. This section describes the Visible, Disable, Key Focus, Blink, Position, Bounds, and Value properties, which are common to all front panel objects. It also introduces some Property Nodes for specific kinds of controls and indicators.

Visible Property



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

Wiring Example—Set the digital control to an invisible state. A Boolean TRUE value makes the control visible, as shown.

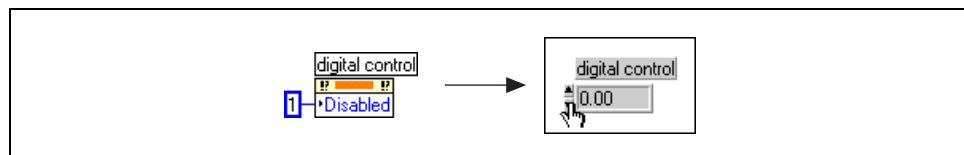


Disabled Property

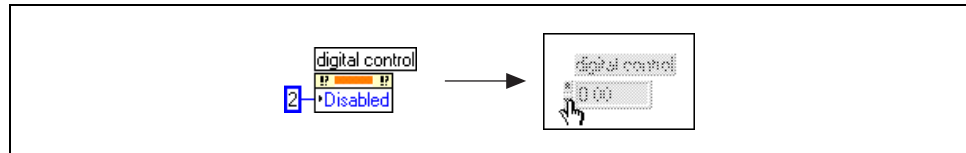


The Disabled property, shown at left, sets or reads the user access status of an object. A value of 0 enables an object so that the user can operate it. A value of 1 disables the object, preventing operation. A value of 2 disables and greys out the object.

Wiring Example—Disable user access to the digital control. The control does *not* change appearance when disabled.



Wiring Example—Disable user access to the digital control and grey it out.

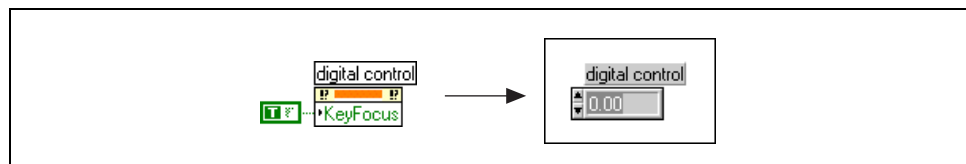


Key Focus Property



The Key Focus property, shown at left, sets 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 with 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).

Wiring Example—Make the digital control the key focus. You can then enter a new value in the control without selecting it with the cursor.

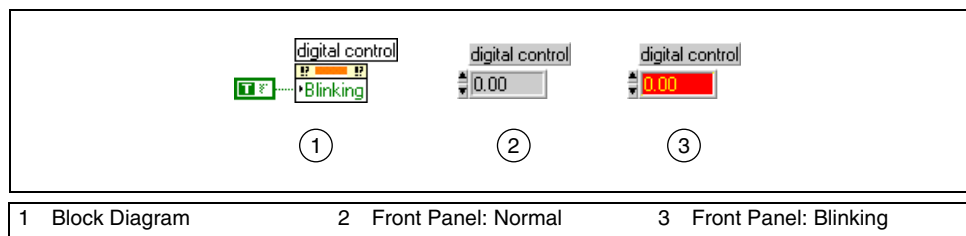


Blinking Property



The Blinking property, shown at left, reads or sets the blink status of an object. By setting this property to TRUE, an object will begin to blink. You can set the blink rate and colors by selecting **Tools»Options** and selecting Front Panel and Colors from the top pull-down menu. When this property is set to FALSE, the object stops blinking.

Wiring Example—Enable blinking for the digital control.

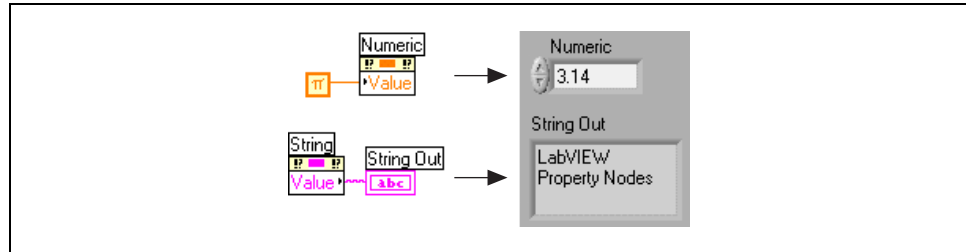


Value Property



The Value property, shown at left, reads or sets the current value of an object. When you set the Value property to write, it writes the wired value to 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. The example

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

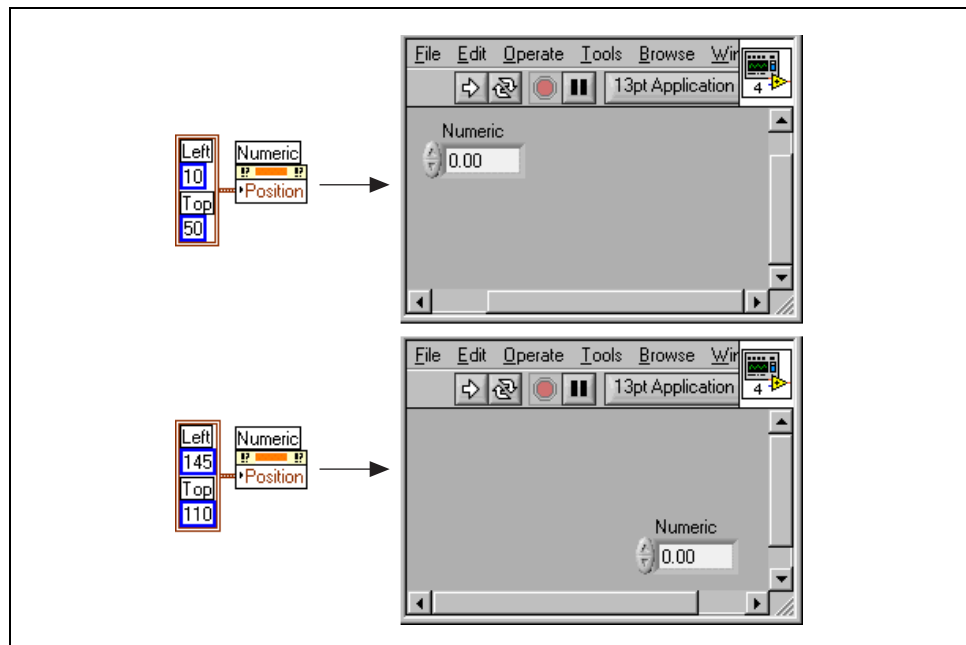


Position Property



The Position property, shown at left, sets or reads 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.

Wiring Example—Make the digital control change its location on the front panel.

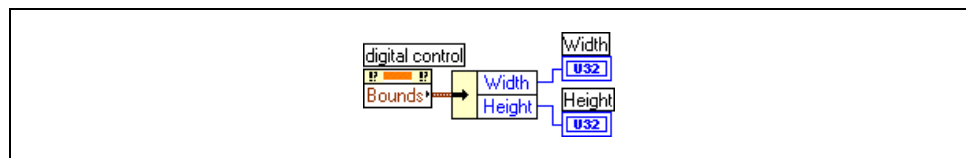


Bounds Property



The Bounds property, shown at left, 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 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.

Wiring Example—Determine the bounds of the digital control.



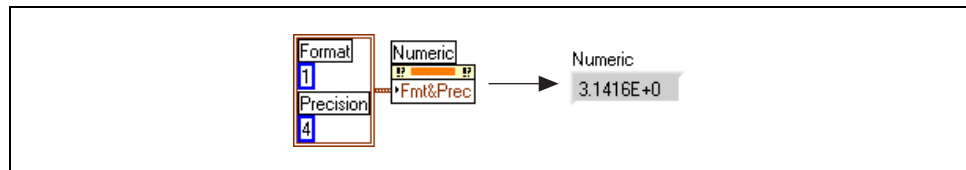
Numeric Property: Format and Precision



The Format and Precision property, shown at left, sets or reads 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

Wiring Example—Set the format of the digital control to scientific notation and the precision to 4.



Boolean Property: Strings [4]

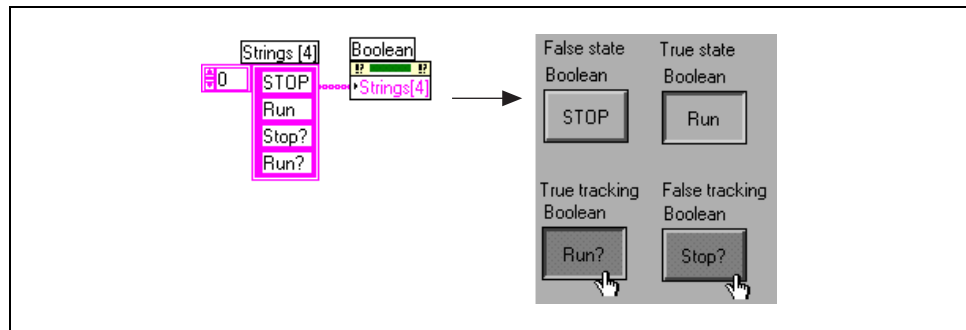


The Strings [4] property, shown at left, sets or reads the labels on a Boolean control. The input is an array of four strings that correspond to the False, True, True Tracking, and False Tracking states.

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. The 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.

Wiring Example—Set the display strings for the Switch control to the string choices Stop, Run, Stop? and Run?



String Property: Display Style



The Display Style property, shown at left, sets or reads the display for a string control or indicator. An unsigned long integer determines the display mode.

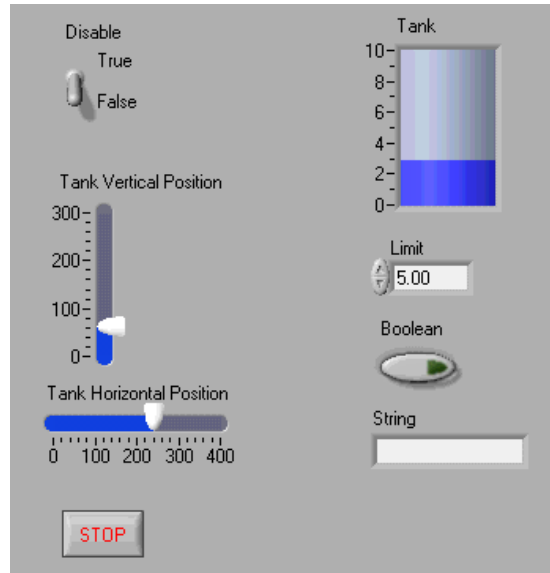
- 0 – Normal Display
- 1 – ‘\’ Codes Display
- 2 – Password Display
- 3 – Hex Display

Exercise 2-5 Property Node Exercise.vi

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

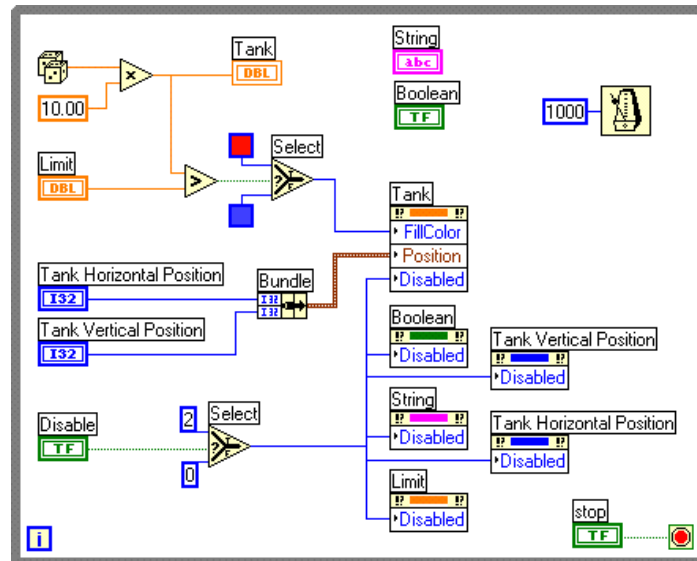
You will build a VI that programmatically changes the position, disabled, and color properties of front panel objects.

Front Panel



1. Open a new VI and build the front panel shown previously.

Block Diagram



2. Open and build the block diagram shown previously using the following components:



- a. Place the While Loop located on the **Functions»Structures** palette on the block diagram. This structures the VI to continue running until the user presses the **Stop** button. Right-click the Conditional terminal and select **Stop If True**.



- b. Place the Wait Until Next ms Multiple function located on the **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**.



- c. Place the Random Number (0-1) function located on the **Functions»Numeric** palette on the block diagram. Creates a random number between zero and one.



- d. Place the Multiply function located on the **Functions»Numeric** palette on the block diagram. 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»Comparison** palette on the block diagram. Compares two values, in this case the random value and the limit value, and returns a True if the random value is greater than the limit. Otherwise it returns a False.



- f. Place the Select function located on the **Functions»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»Numeric»Additional Numeric Constants** palette on the block diagram. This constant is used to color the panel objects through their Property Node. You will need two of these constants. Use the operating tool to select the color and make one red and the other blue.





- h. Create the Tank Property Node on the block diagram. To create this node, right-click the **Tank** terminal and select **Create»Property Node** from the shortcut menu. Resize this node by dragging the corner with the positioning tool to show three terminals. Select the properties shown by right-clicking each terminal and selecting the item from the **Properties** menu. Right-click this node and select **Change All To Write** from the menu.

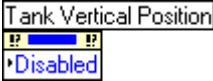


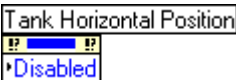
- i. Place the Bundle function located on the **Functions»Cluster** palette on the block diagram. This function clusters together the left and top positions into the Position property for the tank.

 j. Create the Boolean Property Node on the block diagram. To create this node, 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 this node and select **Change To Write** from the shortcut menu.

 k. Create the String Property Node on the block diagram. To create this node, 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 this node and select **Change To Write** from the menu.

 l. Create the Limit Property Node on the block diagram. To create this node, 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 this node and select **Change To Write** from the menu.

 m. Create the Tank Vertical Position Property Node on the block diagram. To create this node, 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 this node and select **Change To Write** from the menu.

 n. 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 this node and select **Change To Write** from the menu.

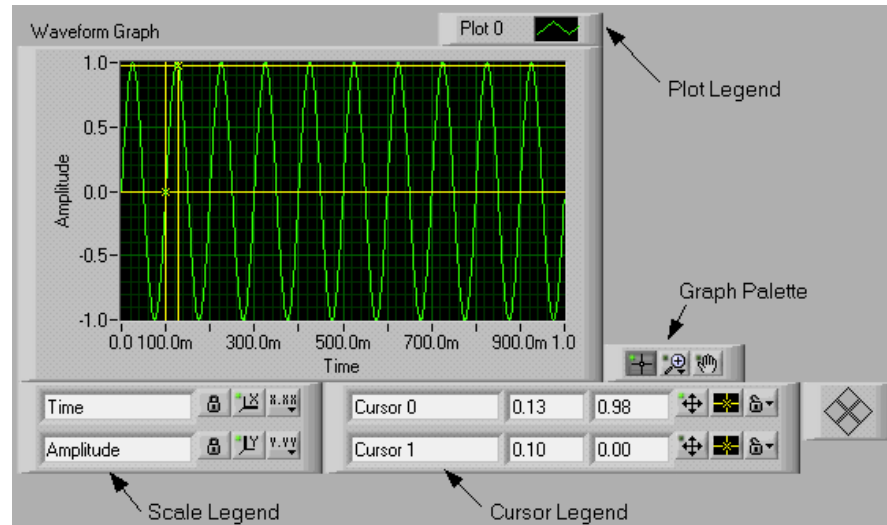
3. Save this VI as `Property Node Exercise.vi`.
4. Return to the front panel and run the VI. Several things should be happening.
5. First, as new random numbers are generated and written to the tank, the fill color is red if the random value is greater than the Limit value and the color is blue if the random value is less than the Limit.
6. The two sliders change the position of the tank on the panel. Move these values and see how the tank moves.

7. 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 grayed out and you cannot change their values.
8. Stop and close this VI when you are finished.

End of Exercise 2-5

D. Graph and Chart Properties

Property Nodes greatly enhance the programmatic flexibility of graphs and charts. You can control most graph and chart features with Property Nodes. Some of these properties are plot color, X and Y scale information, visibility of the legends and palette, size of the plotting area, and cursors.



You have many scale options for graphs and charts. With Property Nodes, you can set or read scale information for the X and Y scales. For each axis, the VI can read or set the minimum, maximum, and increment values.

You also can use Property Nodes to programmatically read or set the plot and background colors, and the X and Y grid colors. You can use this capability to set plot or background colors depending on program conditions such as out-of-range or error values.

To change the size of a graph or chart, use the **Plot Area»Size** property. Thus, if you want to have a particular 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 a VI's panel.

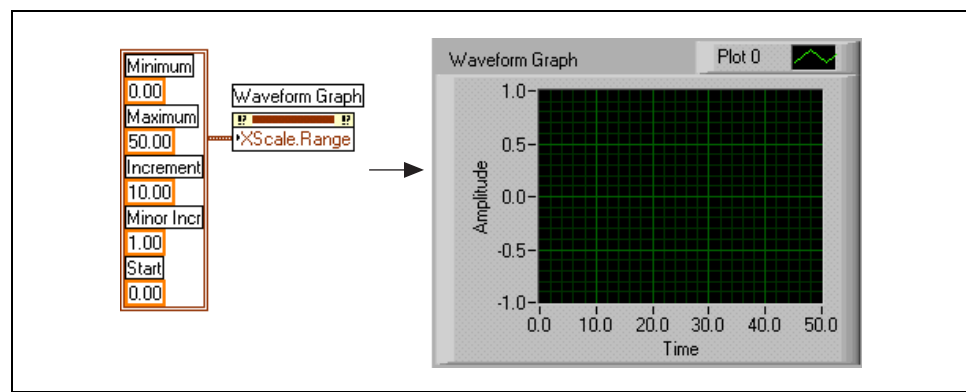
Using Property Nodes, you have access to the information that cursors provide on graphs. You use cursors to move a graphic selector on the plot. A cursor can be locked to the plot or can float on the plotting surface. Values from each cursor are returned to an optional display on the front panel. Property Nodes present a means to programmatically set or read cursor position on the graph, allowing user input from cursors in the VI.

X (or Y) Range Property



The X (or Y) Range Property, shown at left, 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: X axis minimum value, X axis maximum value, major and minor increments between the X 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 will select an alternate increment value.

Wiring Example—Set the X axis range to 0 to 50 with major axis increments of 10, minor increments of 1, and a start value of zero.

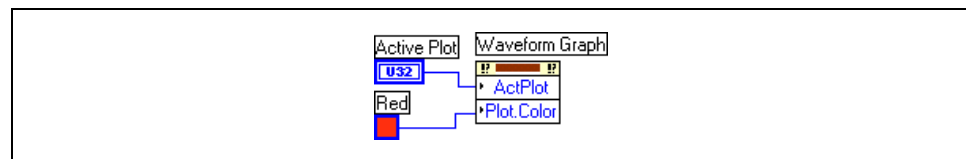


Active Plot and Plot Color Properties



The properties shown at left 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.

Wiring Example—Set the color of the active plot using a Color Box Constant (set to Red in this example). When selecting the active plot, the Active Plot terminal must precede (appear previous) the Plot Color terminal.



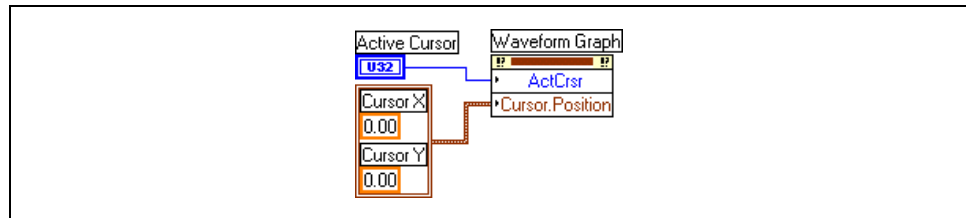
Active Cursor, Cursor Position, and Cursor Index Properties



The properties shown at left 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**. Cursor Index accepts an integer corresponding to an element in the array (plot). To access the Cursor Index property, select **Cursor»Cursor Index**.

Wiring Example—Place 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.

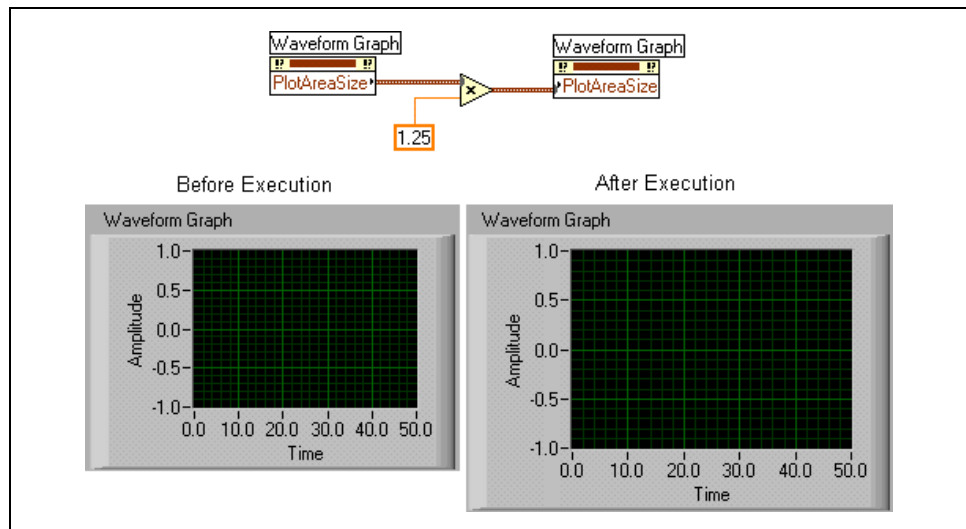


Plot Area»Size Property



To read or change the size of a graph or chart, send new Width and Height values to the Plot Area Size property. The Width and Height are in units of screen pixels.

Wiring Example—Resize a graph to increase its width and height by 25 percent.



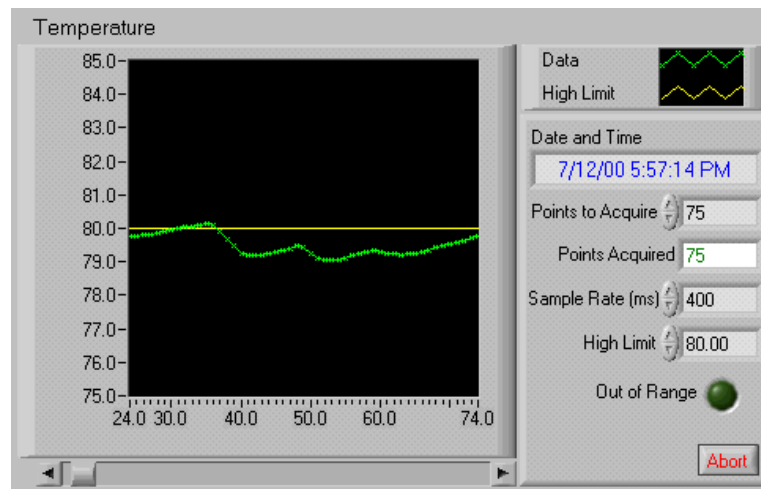
Exercise 2-6 Temperature Limit.vi

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:

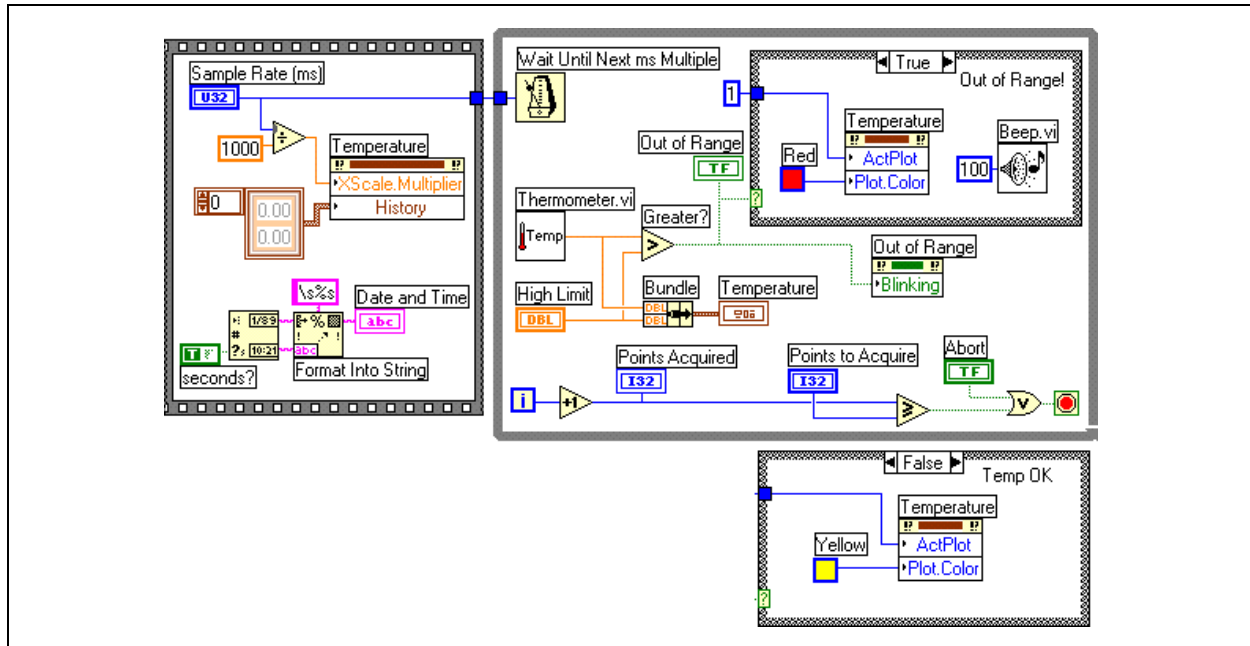
- Set the delta X value of the chart to the sample rate in seconds.
- Clear the waveform chart so it initially contains no data.
- Change the color of a plot if the data exceed a certain value.
- Make an alarm indicator blink if the data exceed a certain value.

Front Panel



1. Open the Temperature Limit VI in the `c:\exercises\LV Basics 2` directory. The front panel and a portion of the block diagram are already built for you.

Block Diagram



1. 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” line red when the temperature exceeds the limit value, and the Out of Range LED should blink.
 - a. In the Sequence structure, create a Property Node for the Temperature chart that has two terminals. One terminal should be the multiplier property, **X Scale»Offset and Multiplier»Multiplier**, and the other terminal should be the History Data property. To clear a waveform chart from the block diagram, send an empty array of data to the History Data property. To do this, right-click the 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. The top terminal should be the Active Plot property, and the bottom terminal should be the Plot Color property, **Plot»Plot Color**. You will need these properties in both cases of the Case structure. The High Limit plot is plot 1, so 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, it should be yellow. Use a Color Box constant to send the color value to this property.
 - c. Right-click the Out of Range indicator to create the Blink Property Node. The indicator should blink when the temperature is greater than the High Limit.

2. Save the VI under the same name.
3. Run the VI to confirm that it behaves correctly. Save and close the VI.

End of Exercise 2-6

Exercise 2-7 Analyze & Present Data.vi

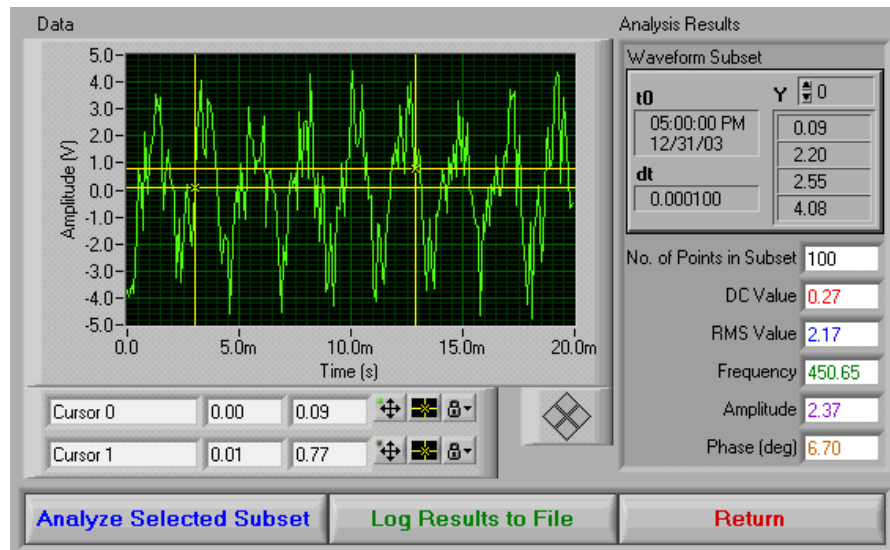
Objective: To use Property Nodes with graph cursors.

You will 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. You will also use the Value property to initialize an indicator.





Note You will use this VI in the project in Lesson 5.

Front Panel



1. Open the Analyze & Present Data VI in the `c:\exercises\LV Basics 2` directory. The front panel for this VI is already built. You will complete the block diagram.

You will use the two cursors shown 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.

In this exercise, use cursors that are locked to the plot. Use Property Nodes to initially set the cursor indices to the beginning and end of the array of data when the VI starts. When the user clicks the **Analyze Selected Subset** button, read the location of each cursor and use this information to find the DC, RMS, frequency, amplitude, and phase values of the subset of data.



Place the Basic Averaged DC-RMS VI located on the **Functions»Analyze»Waveform Measurements** palette on the block diagram. This VI takes the input waveform and calculates the DC and RMS values.



Place the Extract Single Tone Information VI located on the **Functions»Analyze»Waveform Measurements** palette on the block diagram. This VI takes the input waveform, calculates the frequency response, and returns the single tone with highest amplitude. LabVIEW returns the amplitude, frequency, and phase values of that single tone.

2. Save the VI under the same name.
3. Run the VI. Move the cursors along the graph to select a subset of data to analyze, and then click the **Analyze Selected Subset** button. The results appear in the Analysis Results cluster. When you have finished, click the **Return** button.
4. Close the VI when you are finished.

End of Exercise 2-7

E. Control References

In the previous two sections you used Property Nodes to programmatically change the characteristics of panel objects. If you are building a large VI that contains many Property Nodes or if you are using the same property for many different controls and indicators, you might want to consider placing the Property Node in a subVI and using Control References to access that node. A Control Reference is a refnum to a specific control. This section shows only one way to use these versatile objects. Refer to the LabVIEW manuals for more information about control references.

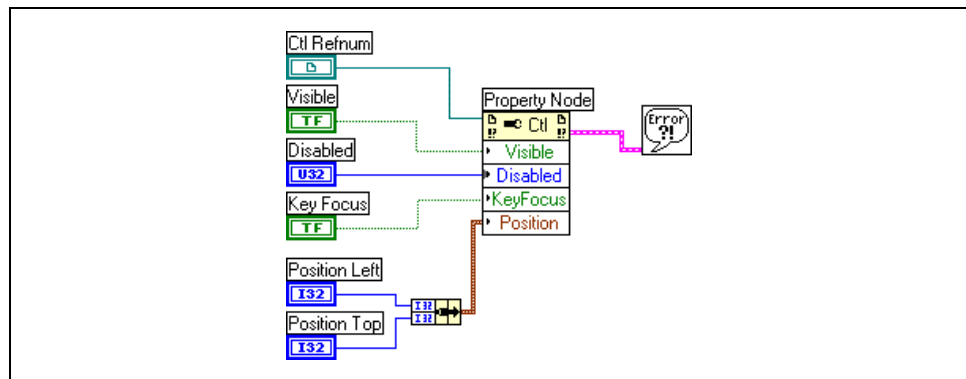
Creating Control References

You create a control reference for a panel object by right-clicking either on the panel object or on the terminal for that object and selecting **Create»Reference** from the shortcut menu.

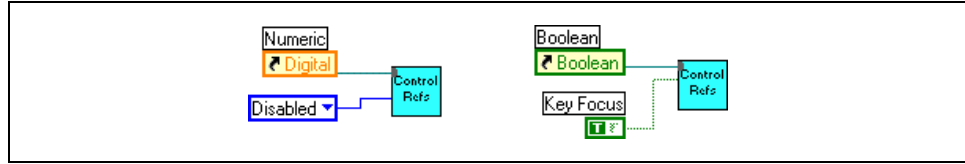
You then wire this Control Reference to a subVI that contains the Property Nodes. However, the subVI must contain a terminal that is a Control Refnum. You create the Control Refnum by selecting **Controls»Refnum** as follows.

Using Control References

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



Property Nodes use error clusters like many of the I/O functions in LabVIEW so you can use the error handlers to keep track of the error conditions. The VI that calls this subVI can use it to change the properties of any type of control. The following example shows the subVI disabling a numeric and setting a Boolean object to be the key focus.



Next you will build a subVI that changes the disabled property for an array of controls using the controls references.

Exercise 2-8 Disable Controls.vi and Control Refs Example.vi

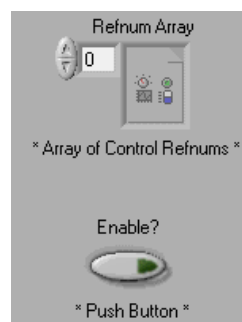
Objective: To build a VI that uses control references to access Property Nodes from a subVI.

You will start with the Property Node Exercise VI you built in Exercise 2-5. You will make a subVI that accesses an array of control references and assigns the Disabled property. Then you will modify the Property Node Exercise VI to use the subVI rather than the original Property Nodes.



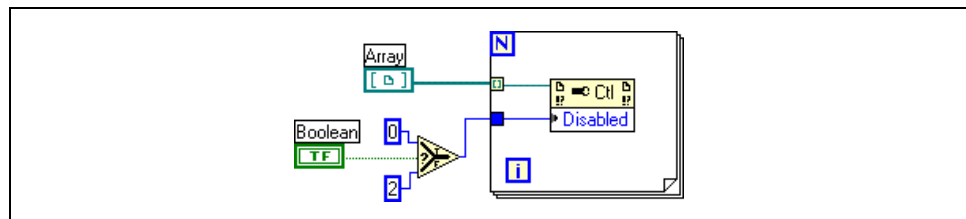
Note You will use this VI in the project in Lesson 5.

Front Panel



1. Open a new VI and build the front panel shown previously.

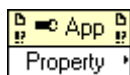
Block Diagram



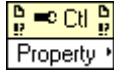
2. Open and build the block diagram shown previously using the following components:



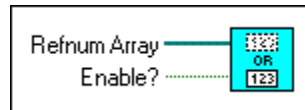
- a. Place a For Loop located on the **Functions»Structures** palette on the block diagram. The For Loop is used to auto-index through the array of control refs so that each refnum and property is handled separately.



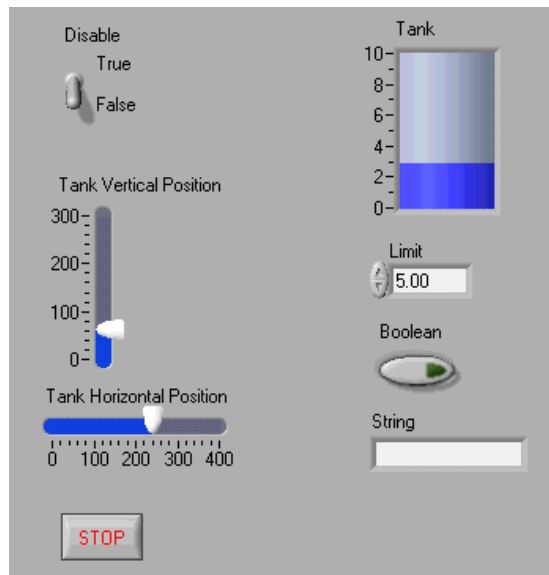
- b. Place a Property Node located on the **Functions»Application Control** palette on the block diagram. You will use this Property Node as a generic control type. When you wire the Refnum Array to the refnum input terminal, the function changes slightly as in the following example:



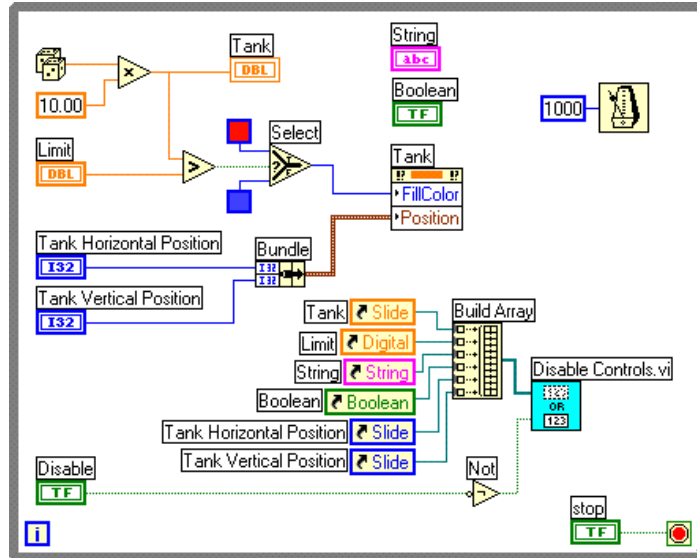
- c. Right-click the Property terminal and select **Properties»Disabled**. Right-click the terminal again and select Change To Write.
 - d. Place the Select function located on the **Functions»Comparison** palette on the block diagram. This function takes a Boolean input and outputs the top value of 0 (enabled) if the Boolean is True and the bottom value of 2 (disabled and grayed out) if the Boolean is False.
 - e. Place a Numeric constant located on the **Functions»Numeric** palette on the block diagram. You will need two of these with a value of 0 and 2 for the Select function.
3. Save this VI as `Disable Controls.vi`.
 4. Return to the front panel. You will now build an icon and connector pane as follows for the VI.



5. Save and close this VI. You will now make a calling VI for this subVI.



6. Open the Property Node Exercise VI you built in Exercise 2-5. You will not modify the panel.
7. Select **File»Save As** and rename this VI `Control Refs Exercise.vi`.



8. Open the block diagram and modify it as shown previously.
 - a. Create Control References for the six controls by right-clicking their terminals and selecting **Create»Reference** from the shortcut menu.
 - b. You use the **Build Array** function located on the **Functions»Array** palette to combine all the control references into an array to pass it to the **Disable Controls** subVI located on the **Functions»Select A VI** palette.
 - c. You also need the **Not** function located on the **Functions»Boolean** palette to invert the value going into the subVI.
9. Save the VI.
10. Return to the panel and run the VI. Notice that when you set the **Disable** switch to **True**, all the controls become grayed out as they did before.
11. Close this VI when you are finished.

End of Exercise 2-8

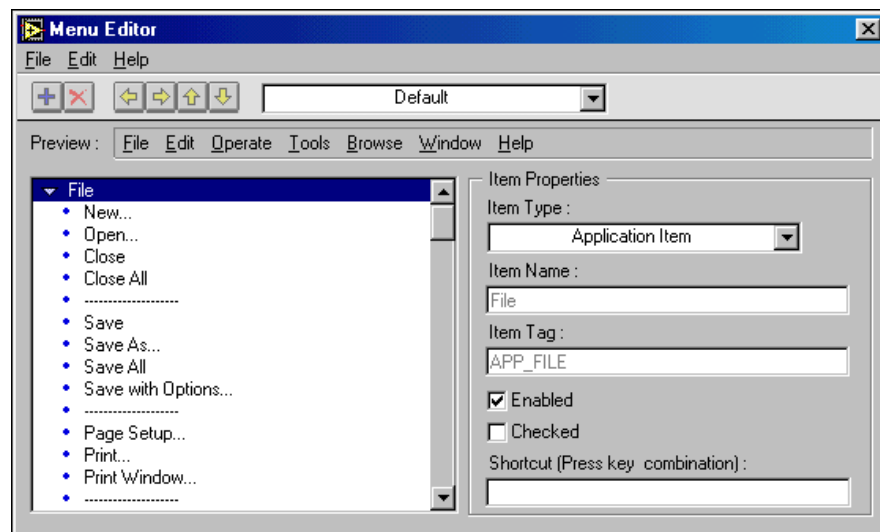
F. LabVIEW Run-Time Menus (Optional)

You can customize the menu bar for every VI you build so that you have a custom pull-down menu while a VI executes. There are two parts to customizing menus—creating menus and handling menus.

You can build custom menus in two ways—statically at the time of editing, or dynamically at run time. You can statically store a menu template in a file called the *run-time menu* (RTM) file. You can associate an RTM file with a VI at the time of editing. When the VI runs, it loads the menu from the associated RTM file. You also can programmatically insert, delete, and modify menu items at run time from the block diagram using several LabVIEW functions. In addition, you can programmatically determine if an option on the menu bar is selected, and programmatically handle the selection. With these features, you can make a custom pull-down menu for your application.

Static Menus

Static menus are stored in RTM files. You can enable, create, or edit an RTM file for a VI by selecting **Edit»Run-Time Menu**. The **Menu Editor** dialog box appears.



On the left side of this dialog box is an overall organization of the menu items. Menu items are classified into three types: User Item, Application Item, or Separator.

A **User Item** can be handled programmatically in the block diagram. It has a *name*, which is the string that appears in the actual menu, and a *tag*, which is a unique case-insensitive string identifier. A tag identifies a **User Item** in the block diagram. For ease in editing, when you enter the name, it is copied to the tag. However, you can always edit the tag to make it different from the

name. For a menu item to be valid, its tag should not be empty. Invalid menu items appear as ???.

An **Application Item** is one that LabVIEW provides. These items are part of the default LabVIEW menu. To select a particular LabVIEW item, click the arrow button next to the Item Name. You can add individual items or entire submenus by using this process. Application Items are handled implicitly by LabVIEW. These item tags do not appear in block diagrams. You cannot alter the name, tag, or other properties of an Application Item. LabVIEW reserves tags starting with APP_ for Application Items.

A **Separator** inserts a separation line in the menu. You cannot set any of the properties for a Separator item.

The **Menu Editor** ensures that the *tag* is unique to a given menu hierarchy by appending numbers when necessary. A User Item can be enabled/disabled or checked/unchecked by setting the respective properties. You can set a shortcut (accelerator) for a User Item by selecting an appropriate key combination.

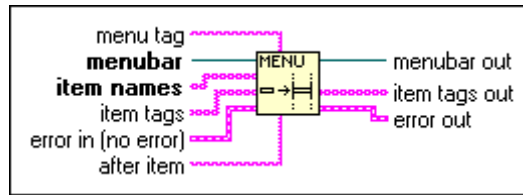
The **Menu Editor** allows you to insert, delete, or reorder menu items in the menu hierarchy. Clicking the + button adds a new menu item. You also can change the type of a menu item by selecting from the **Item Type** ring. In addition, you can reorder the menu items and create submenus using the arrow buttons. Finally, clicking the **Delete Item** button deletes the selected menu item.

The Preview portion of the **Menu Editor** provides an up-to-date view of the run-time menu. On the **Menu Editor** pull-down menu, the **Open**, **New**, **Save**, and **Save As** buttons allow you to load and save RTM files. Once you close the **Menu Editor**, you will have the option of updating the VIs run-time menu with the menu you edited.

Dynamic Menus

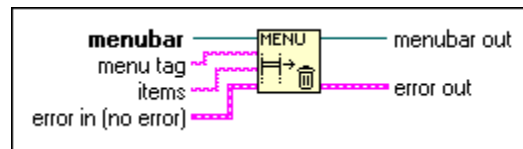
While a VI executes, you can change items in the VI menu bar by using the menu management functions described in the following sections. All of these functions are located under **Functions»Application Control»Menu**. All of these functions operate on a refnum for the menu retrieved using the Current VI's Menu function.

Insert Menu Items



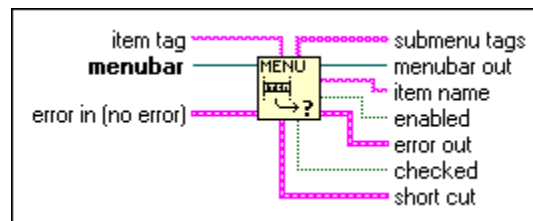
Inserts a menu item (or an array of menu items) identified by **item names** or **item tags** into a menu (menu bar or submenu). The position of the item is specified either through the **after item** parameter or through a combination of the **menu tag** and **after item** pair.

Delete Menu Items



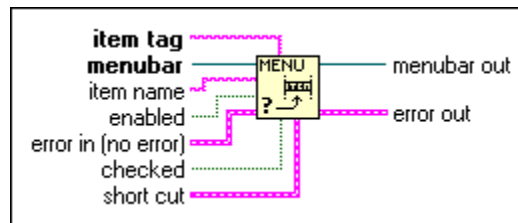
Deletes a menu item (or an array of menu items) identified by **items** from a menu identified by **menu tag** (from the menu bar, if **menu tag** is not specified). **Items** can be a tag, array of tags, position number, or an array of position numbers.

Get Menu Item Info



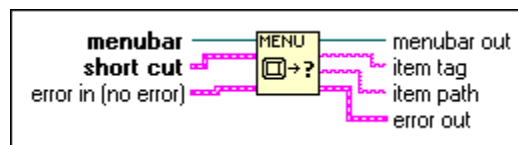
Returns properties of the menu item specified through **item tag** (or the **menubar** if the **item tag** is unspecified). Item properties include **item name**, **enabled**, **checked**, and **shortcut**. If the item has a submenu attached, its item tags are returned in **submenu tags**.

Set Menu Item Info



Sets properties of the menu item specified through item tag. Item properties include item name, enabled, checked, and shortcut. Unwired properties remain unchanged.

Get Menu Shortcut Info



Returns the menu item that is accessible through a given shortcut.

Menu Selection Handling

Menu Selection functions handle your menu selections. The menus might have been built statically in VI Properties or dynamically during run time. To set up the block diagram to handle menu selections, acquire control over the menu selection process with the Get Menu Selection function. While the VI controls the menu selection, it waits for a selected menu item using the same function, Get Menu Selection. All the LabVIEW menu items are implicitly handled by LabVIEW—only the user menu selections are obtained through Get Menu Selection.

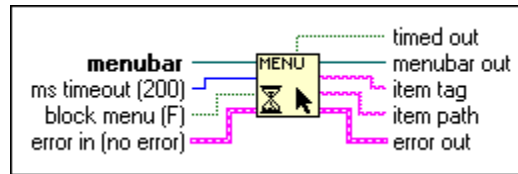
Once you select an item, you cannot select another item until the Get Menu Selection function reads the first item. Under such conditions, Get Menu Selection is invoked under block menu mode, wherein menu tracking is blocked out after a selection is read. The menu is enabled after you process the selection using the Enable Menu Tracking function.

Current VI's Menubar



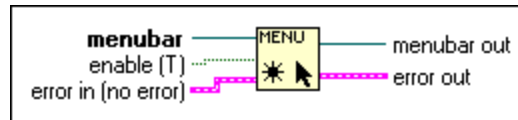
Returns the **menubar** refnum of the current VI. This function must be executed before the other menu handling functions are invoked.

Get Menu Selection



Returns the **item tag** of the last selected menu item, optionally waiting for a period of time specified by **timeout**. **Item path** is a string describing the position of the item in the menu hierarchy. Menu selection is blocked after an item is read if **block menu** is set to TRUE.

Enable Menu Tracking



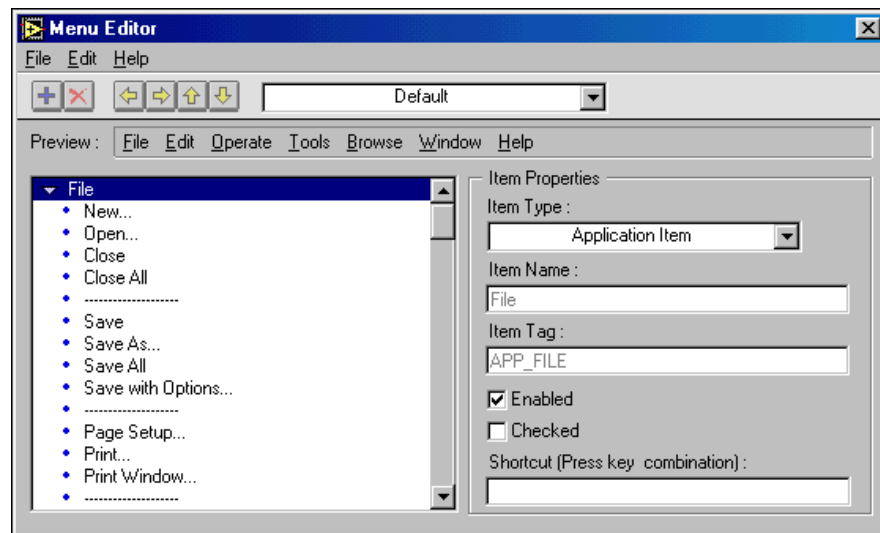
This function enables or disables the tracking of menus. Once a menu is blocked using the Get Menu Selection function, Enable Menu Tracking must be executed to re-enable the pull-down menu.

Exercise 2-9 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.

1. Open the Pull-down Menu VI located in the `c:\exercises\LV Basics 2` directory. The front panel and block diagram are partially complete. You will build a custom run-time menu for this VI using the **Menu Editor**, and complete the block diagram so that you can access this menu.
2. Select **Edit»Run-Time Menu** to display the **Menu Editor** dialog box.

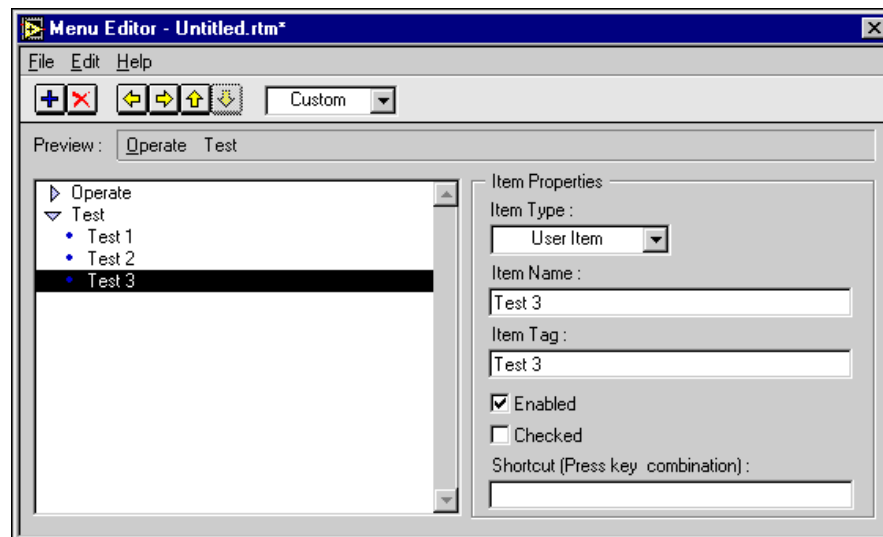


The current run-time menu for the application is the LabVIEW default menu. In the next several steps, you will replace that menu with a custom list of selections.

3. Change the top ring control in the **Menu Editor** from **Default** to **Custom**. The menu listed on the left portion of the dialog box should be replaced with a ???, representing a single unnamed item.
4. In the **Item Type** ring control, select **Application Item»Operate»Entire Menu**. The LabVIEW **Operate** menu should be added to the custom menu. Default LabVIEW options, as well as selections you create, can be added to a custom menu.

Take a moment to navigate the **Operate** menu in the editor. Notice that you can select various items and collapse submenus using the triangle icons. As you select individual items in the menu, their Item Name and Item Tag appear in the Item Properties box 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.

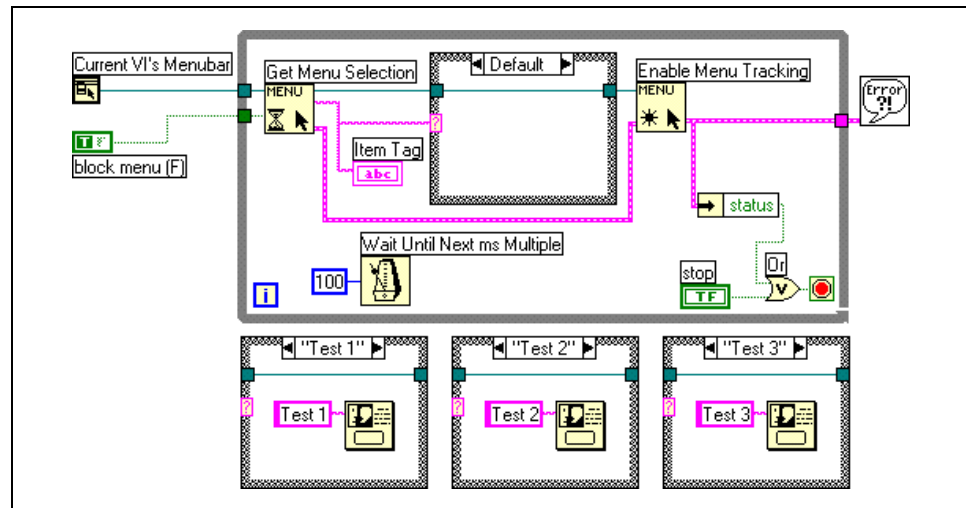
5. Click the **+** button in the **Menu Editor** toolbar. A new unnamed item, ???, appears in the menu list. With this item highlighted, enter `Test` into the **Item Name** property. This menu item now has an item name and tag of `Test`.
6. Click the **+** button again to add another entry under the `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.
7. Add two more subitems under the `Test` submenu called `Test 2` and `Test 3`. The **Menu Editor** dialog box should now resemble the following:



In the Preview area of the **Menu Editor**, you can see how the custom menu will behave during run time.

8. Select **File»Save** from the **Menu Editor** dialog box. Save the run-time menu as `Menu Exercise.rtm` in the `c:\exercises\LV Basics 2` directory. Then 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 pull-down menu which will be invoked while the VI executes.

Block Diagram



9. Change to the block diagram of the VI and complete it as shown previously.



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



b. Place the Get Menu Selection function located on the **Functions»Application Control»Menu** palette on the block diagram. Each time the While Loop executes, the Get Menu Selection function will return 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 is executed.



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

10. Save the VI and run it. When the VI executes, the custom run-time menu appears on the front panel. If you select one of the items in the **Test** pull-down 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 pull-down menu item, you find that the menu is disabled (this is caused by the block menu parameter of the Get Menu Selection function). If you click **OK** 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** pull-down menu

do not show up 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 **Stop** on the VI's front panel to halt program execution.

- Using the **Menu Editor**, you can assign shortcut keys to User items you create. Assign the keyboard shortcuts to the test options according to the following table:

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>

In addition to setting the previous shortcuts, on a Windows platform you 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.

- 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.
- Close the VI.

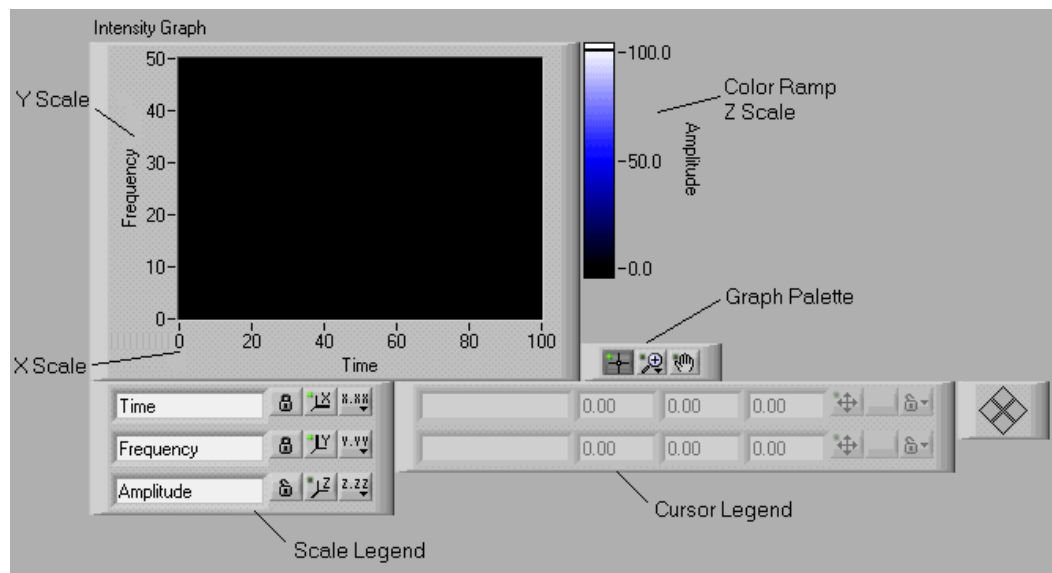
End of Exercise 2-9

G. Intensity Plots

Intensity plots are extremely useful for displaying patterned data. For example, the plots work well for displaying terrain, where the magnitude represents altitude. Also, you can demonstrate temperature patterns with intensity plots. Like the waveform graph and chart, the intensity chart features a scrolling display while the intensity graph features a fixed display. These displays accept a block of data and map each value to an associated color, with a maximum of 256 accessible colors.

Intensity Plot Options

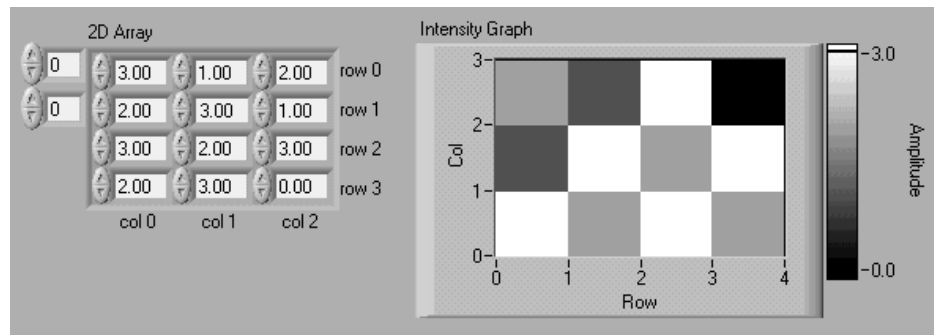
The displays of intensity charts and graphs use many of the same options as their waveform counterparts. The following front panel shows many of the intensity graph options. Intensity plots also provide options unavailable in other charts and graphs. Because they display a third value (color), a color ramp is accessible. You use these options to set and display the color mapping scheme. The intensity graph also adds a third value, a z-value, to the cursor display.



To change the color associated with a specific intensity value, right-click the marker appearing next to the color ramp and select **Marker Color** from the menu. The color palette appears, from which you select a color to associate with that particular numeric value. To add more values to the color ramp, right-click it and select **Add Marker**. You can drag the tick mark that appears to the appropriate location on the color ramp, or select the text next to the tick mark with the Labeling tool and type in the location of the marker. After you have placed the marker at the appropriate location on the ramp, right-click it and select **Marker Color**.

Intensity Plot Data Types

The intensity chart and intensity graph accept a 2D array of numbers. The location of each element in the array (its row and column indices) maps it to the X and Y value on the chart or graph. The magnitude of each array element maps to a corresponding color. The example following shows a 4×3 array plotted on an intensity graph. Notice how the Intensity Graph transposes the array elements.

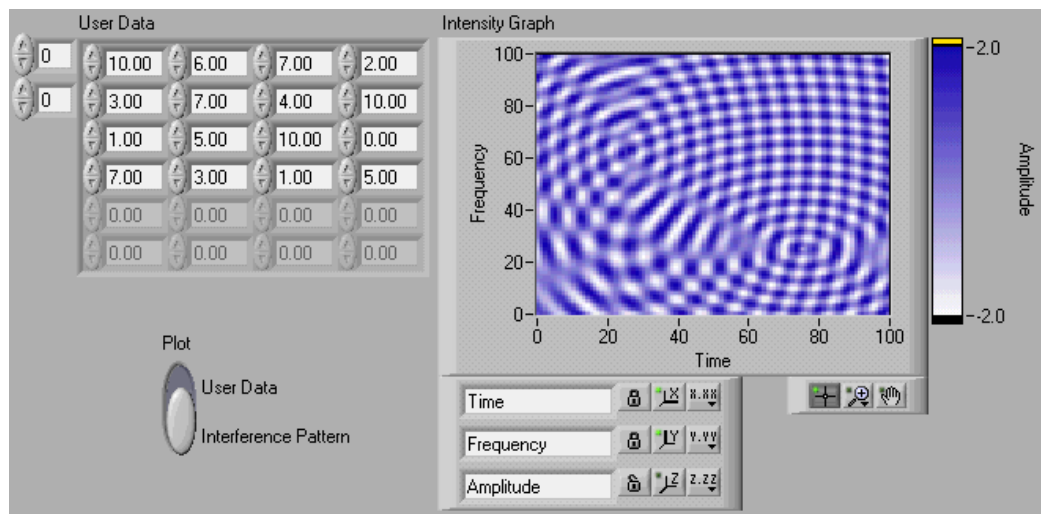


Exercise 2-10 Intensity Graph Example.vi

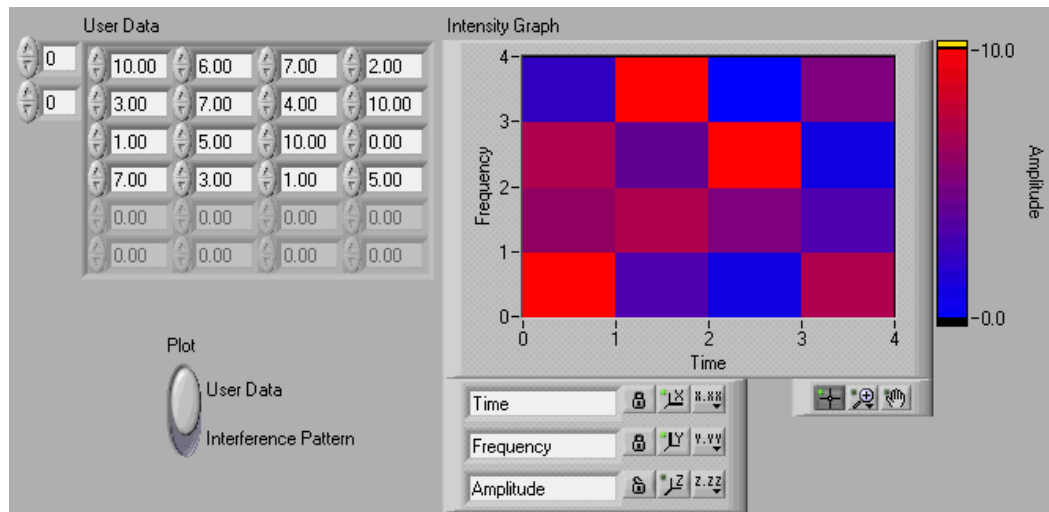
Objective: To use an intensity graph.

You will use a VI that displays a wave interference pattern on an intensity graph. You will also use the VI to plot a 2D array of data on the graph.

1. Open and run the Intensity Graph Example VI in the `c:\exercises\LV Basics 2` directory. By default, the VI plots an interference waveform. A Property Node on the block diagram defines the color range used in the intensity graph. You can change the color range by opening the block diagram and modifying the Color Array constant.



2. Change the Plot switch on the front panel to User Data and enter values between 0.0 and 10.0 in the User Data array control. Run the VI. Notice how the magnitude of each element is mapped to the intensity graph.



3. Close the VI. Do not save changes.

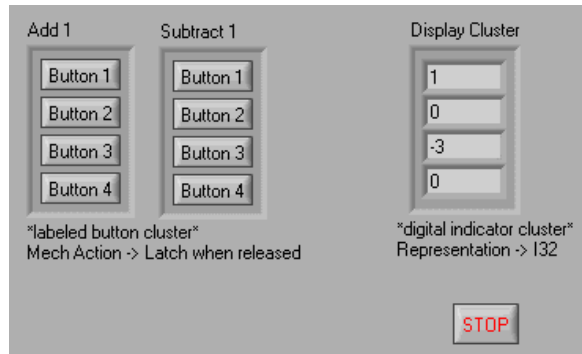
End of Exercise 2-10

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.
- 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 (**Functions»Cluster or Functions»Array** palette) converts a cluster to a 1D array.
- Property Nodes are powerful tools for expanding user interface capabilities. You use Property Nodes to programmatically manipulate the appearance and functional characteristics of 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, so you can configure properties from several locations in the block diagram.
- You can set or read properties such as user access (enable, disable, gray out), 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.
- 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»Application Control»Menu** palette.
- You can use intensity charts and graphs to plot three-dimensional data. The third dimension is represented by different colors corresponding to a color mapping that you define. Intensity charts and graphs are commonly used in conjunction with spectrum analysis, temperature display, and image processing.

Additional Exercises

- 2-11 Modify Exercise 2-4 by adding a cluster of four labeled buttons. Each time you click a button in the new cluster, decrement the display by one. Use the front panel as follows to get started. Save the VI as `Cluster Example 2.vi`.

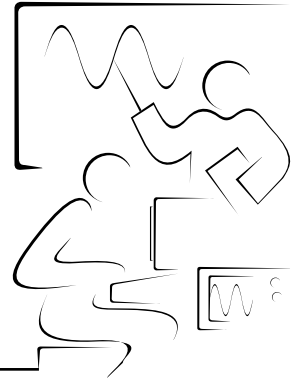


- 2-12 Build a VI that manipulates a button control on the front panel. The button should control the VI's execution (that is, terminate the While Loop). Use a Dialog Ring to select the following options: **Enable** and **Show** button, **Disable** button, **Grey Out** button, and **Hide** button. Use 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. Save the VI as `Button Input.vi` when you are finished.
- 2-13 Open the Analyze & Present Data VI you developed in Exercise 2-7 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 grayed out.

Notes

Lesson 3

Data Management Techniques



This lesson describes how you can organize the block diagrams of your VIs based upon how you transfer data from one place to another. First, you will be reminded of how LabVIEW runs a VI. Then you will learn how to use local and global data to pass data within and between VIs and tips for using those data objects efficiently. Last, you will learn how to use DataSocket to pass data between VIs and computers.

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 Locals and Globals
- E. How to use DataSocket

A. Data Management Techniques in LabVIEW

The previous lesson describes how you build the user interface, or front panel, for a VI. This lesson concentrates on block diagram issues such as passing data between nodes and controlling execution order.

As you already know, the principle that governs how a LabVIEW program executes is called data flow. The rules of data flow are:

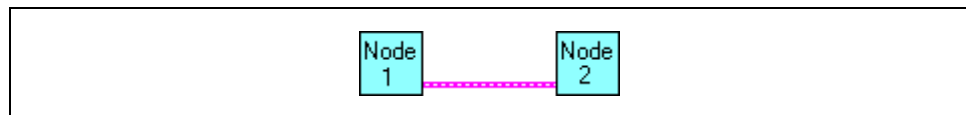
- A node executes only when data is available at all its input terminals.
- A node supplies data to all of its output terminals when it finishes executing.
- The data flows through wires instantaneously.

Therefore, a block diagram does not execute top to bottom or left to right, but according to data flow rules. This contrasts with the control flow method of executing a conventional program in which instructions execute in the sequence in which you write them.

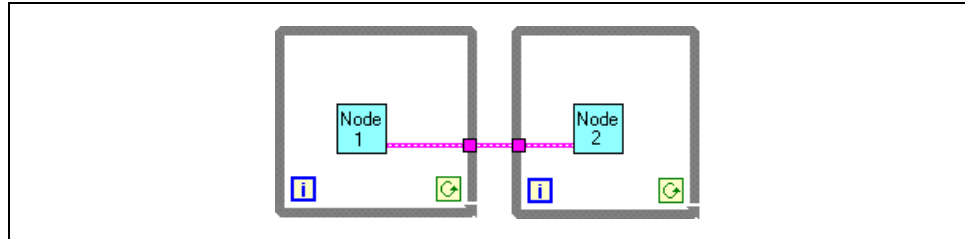
Wires are the key to understanding how data flow works in LabVIEW. When wires directly connect one node to the next, the order of execution is defined. If no wire connects two nodes, then those nodes can run concurrently. If you require those two nodes to run in order, then you either use a sequence structure or, if the nodes are subVIs, program data dependencies such as the error clusters to control program execution.

When You Cannot Use Wires

Wires are the most efficient way to pass data from one node to the next when using data flow programming. On the block diagram, data is transferred from one function or VI to another by connecting them with a wire. The data then flows from the producer to the consumer as shown below.



This approach does not work if you need to exchange data between block diagrams that run in parallel. Parallel block diagrams can be in two parallel loops on the same block diagram or in two VIs that are called without any data flow dependency. Consider the following block diagram.



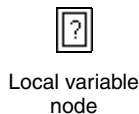
This block diagram does not run the two loops in parallel because of the wire passing between the two subVIs. The wire creates a data dependency where the second loop does not start until the first loop is finished and passes the data through its tunnel. The only way to make the two loops run concurrently is to remove the wire connection. However, then the subVIs cannot pass data between each other unless you use another technique of data management. This lesson describes local variables, global variables, and DataSocket as methods to pass data between parts of the same block diagram and between different block diagrams.

B. Local Variables

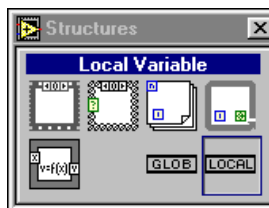
Up until now, you have read data from or updated a front panel object using its terminal on the block diagram. However, a front panel object has only one terminal on the block diagram, and you might need to update or read a front panel object from several locations on the block diagram. Using local variables, you can access front panel objects in several places and pass data between structures that cannot be connected by a wire.

Creating Local Variables

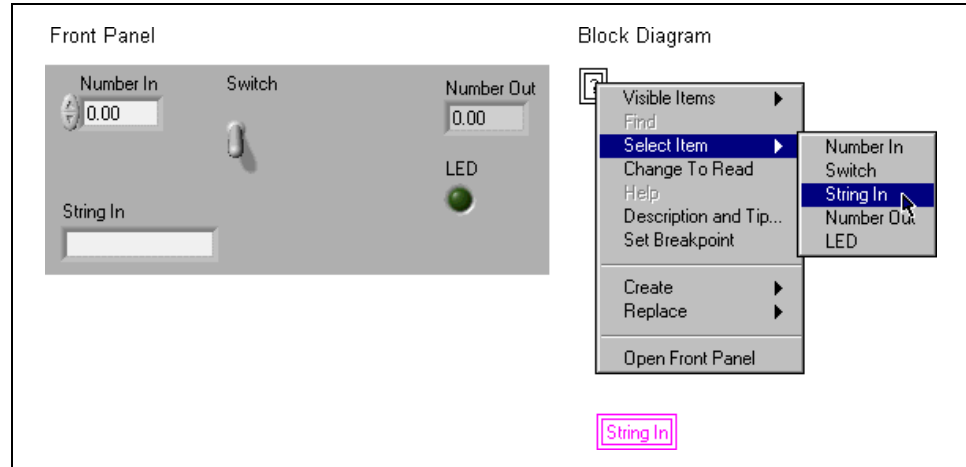
There are two ways to create local variables on the block diagram. If you have already created a front panel object, you can create a local variable by right-clicking the object or its terminal and selecting **Create»Local Variable** from the pop-up menu. You can use this method on the front panel or the block diagram. A local variable icon for the front panel object appears next to the terminal on the block diagram. When you right-click a terminal to create a local variable, the local variable refers to the object you right-clicked to create the icon.



You also can select **Local Variable** from the **Structures** palette. A local variable node, shown at left, appears on the block diagram. You can select the front panel object you want to access by right-clicking the variable node and selecting an object from the **Select Item** menu. This menu lists the owned labels for the front panel controls and indicators. Thus, always label your front panel controls and indicators with descriptive names, using owned labels.



For example, if the first object you create on the front panel is labeled number in, the local variable icon appears as shown at left. After you place the local variable on the block diagram, you can select the appropriate front panel object by either clicking the variable using the Operating tool, or right-clicking it and selecting the front panel object from the **Select Item** menu.



Read Locals and Write Locals

You can either read data from or write data to a local variable. After you place the local variable on the block diagram, you must decide how to use it.

By default, a local variable assumes that it will *receive* data. Thus, this kind of local variable acts like an indicator and is a *write local*. When you write new data into the local variable, the associated front panel control or indicator updates to contain the new data.

You can change the configuration of a local variable so that it acts as a data source, or a *read local*. To do this, right-click the variable and select **Change To Read**. On the block diagram, a read local icon behaves just like a control. When this node executes on the block diagram, your program reads the data in the associated front panel control or indicator.

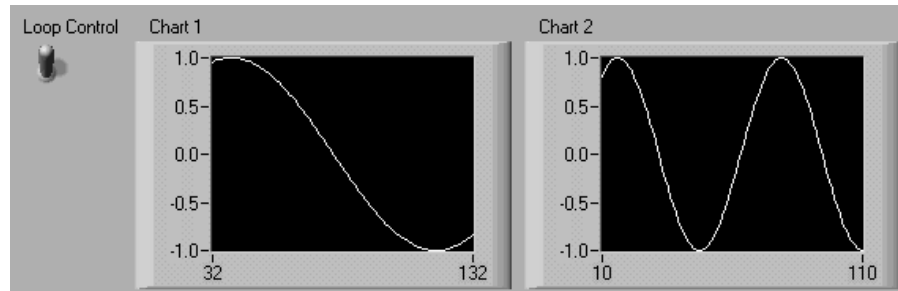
To change a read local to a write local, right-click the variable and select **Change To Write**. The local variable will change its data direction so that it receives data instead of providing data.

On the block diagram, you can visually distinguish read locals from write locals just as you distinguish controls from indicators. A read local has a thick border, emphasizing that it is a data source, similar to a control. A write local has a thin border, because it acts like a data sink, similar to an indicator. In the following figure, both local variables refer to the same item on the front panel.



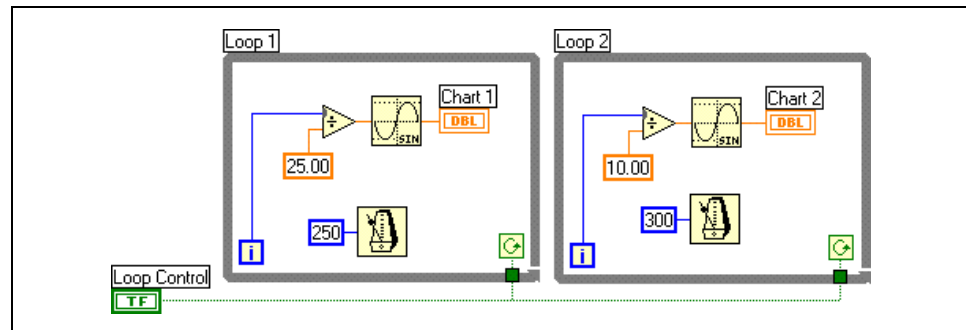
Local Variable Example

As an example of when you might need a local variable, consider how you would use a single front panel switch to control two parallel While Loops. Parallel While Loops are not connected by a wire, and they execute simultaneously. First, we will study two unsuccessful methods using one terminal for the switch on the block diagram (Methods 1 and 2). Then, we will show how a local variable accomplishes the task (Method 3). The front panel of this example VI appears below.



Method 1 (Incorrect)

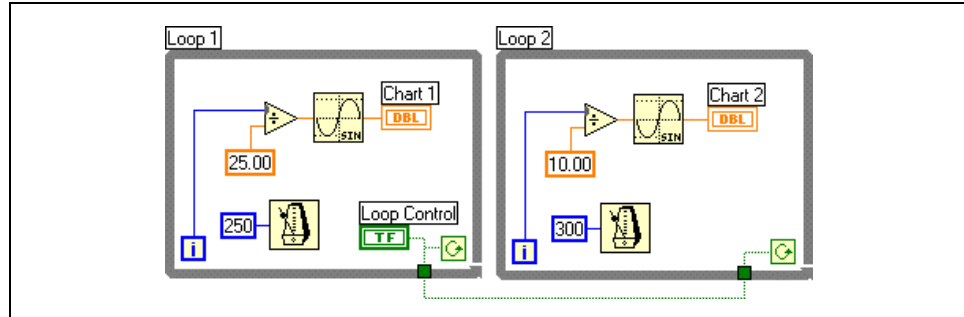
As a first attempt, we place the Loop Control terminal outside of both loops and wire it to each conditional terminal. In this case, the Loop Control terminal is read only once, before either While Loop begins executing. Recall that this happens because LabVIEW is a *dataflow* language, and the status of the Boolean control is a data *input* to both loops. If a TRUE is passed into 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. This solution does not work.



Method 2 (Incorrect)

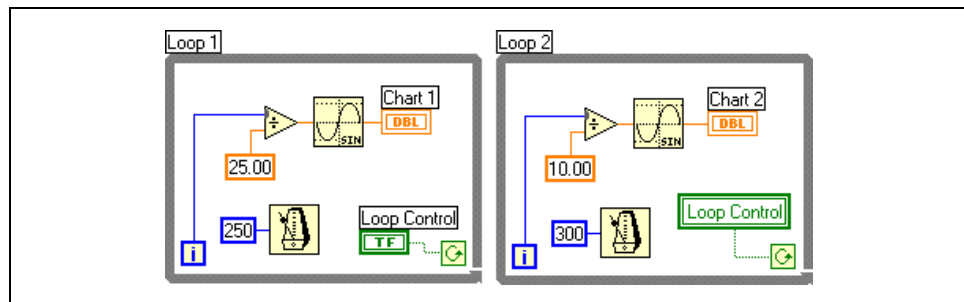
Now we move the Loop Control terminal inside Loop 1 so that it is read in each iteration of Loop 1. Although Loop 1 terminates properly, there is a problem with this approach. Loop 2 does not execute until it receives all its data inputs. Remember that Loop 1 does not pass data out of the loop until the loop stops. Thus, 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)

In this example, Loop 1 is again controlled by the Loop Control switch, but this time, Loop 2 reads a local variable associated with the switch. 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 Loop 2's conditional terminal. Thus, the While Loops run in parallel and terminate simultaneously when the single front panel switch is turned off.



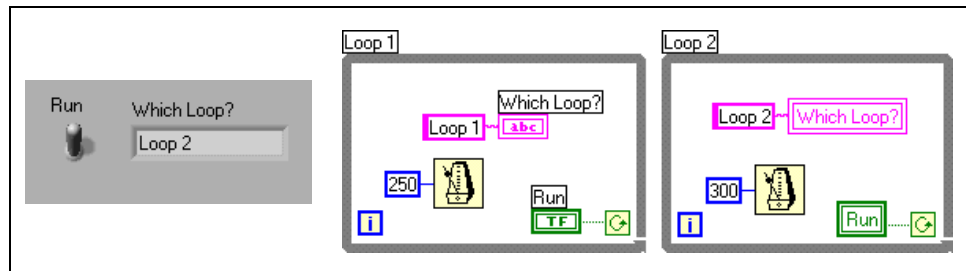
This simple example demonstrates the need for local variables. As previously shown, using the local variable gives access to a single front panel object from several locations on the block diagram. Local variables are also necessary when you cannot accomplish your goal using wires to carry the data.

Thus far, you have learned that you can read input data from controls and send results to an indicator. But, for example, what if you want to determine which parameters were used to run a VI previously and you want to place those values in controls for the users to modify? How can you write those values into a control?

You cannot do this with standard controls and indicators. Using local variables, you can overcome this limitation. You can update a control from the block diagram. Also, you can have any number of local variable references for a given front panel control, with some in write mode and others in read mode. With a local variable reference, you can access a front panel object as both an input and an output.

To understand this concept, we will look at an example that shows another use of local variables. Below is a single string indicator. Suppose you want to update that indicator to display the loop that is currently executing. Without a local variable, there is no way to accomplish this task. 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. Although this example is simple, it shows how an indicator can be updated from two separate locations on the block diagram.



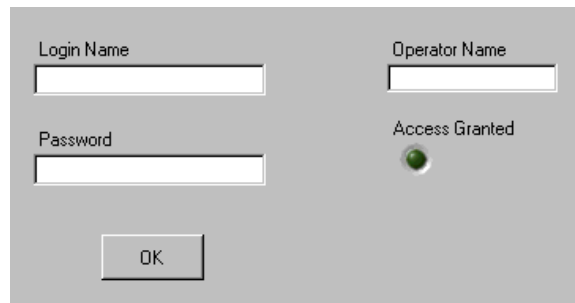
Exercise 3-1 Login VI

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



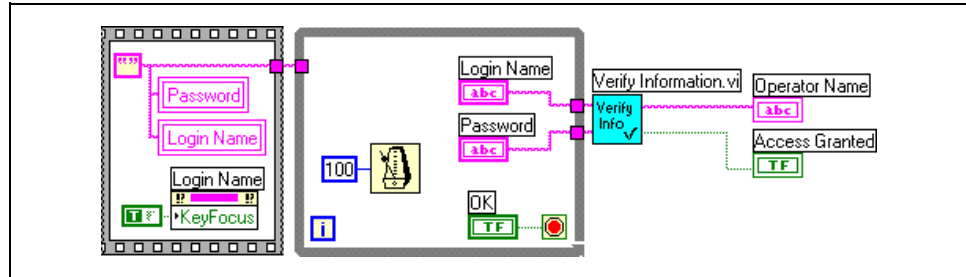
Note Use this VI in the project in Lesson 5.

Front Panel



1. Open the Login VI in the `c:\exercises\LV Basics 2` directory. The front panel of the VI is already created. Finish building the block diagram.

Block Diagram



1. Complete the 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 setup ensures that both local variables are updated *before* the While Loop starts.

Login Name

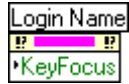
- a. **Login Name** local variable set to write local—Resets the login name to an empty string. To create this local variable, right-click the **Login Name** terminal and select **Create»Local Variable** from the pop-up menu.

Password

- b. **Password** local variable set to write local—Resets the password string to an empty string. To create this local variable, right-click the **Password** terminal and select **Create»Local Variable** from the pop-up menu.



- c. **Empty String** constant, available on the **Functions»String** palette—Passes string values to the Login Name and Password local variables.



- d. **Login Name** Property Node—To create this node, right-click the **Login Name** terminal and select **Create»Property Node**. Change the property to key focus and wire a TRUE Boolean constant to it.

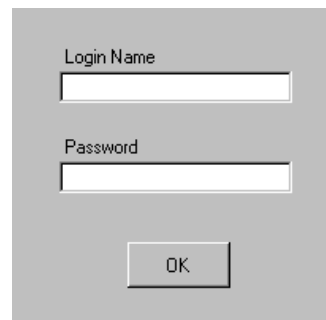


- e. Place the Verify Information VI on the block diagram. To select this VI, select **Functions»Select a VI**, navigate to `c:\exercises\LV Basics 2`, double-click the Verify Information VI, which you built in Exercise 1-2, and place the VI on the block diagram. This VI uses the name and password and checks 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**.

2. Save the VI under the same name.
3. Return to the front panel and run the VI. Notice that the Login Name and Password controls reset to empty strings when you start the VI.
4. Now resize the front panel to show only the necessary objects, as shown:



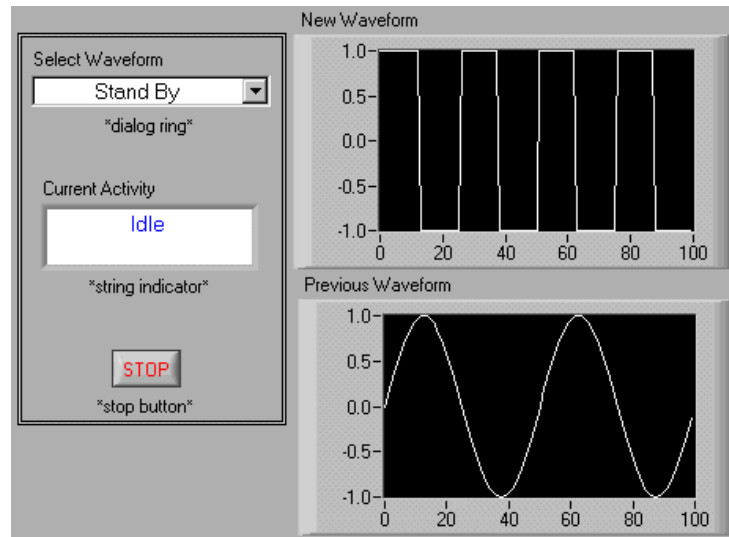
5. Save and close the VI.

End of Exercise 3-1

Exercise 3-2 Select and Plot Waveform VI (Optional)

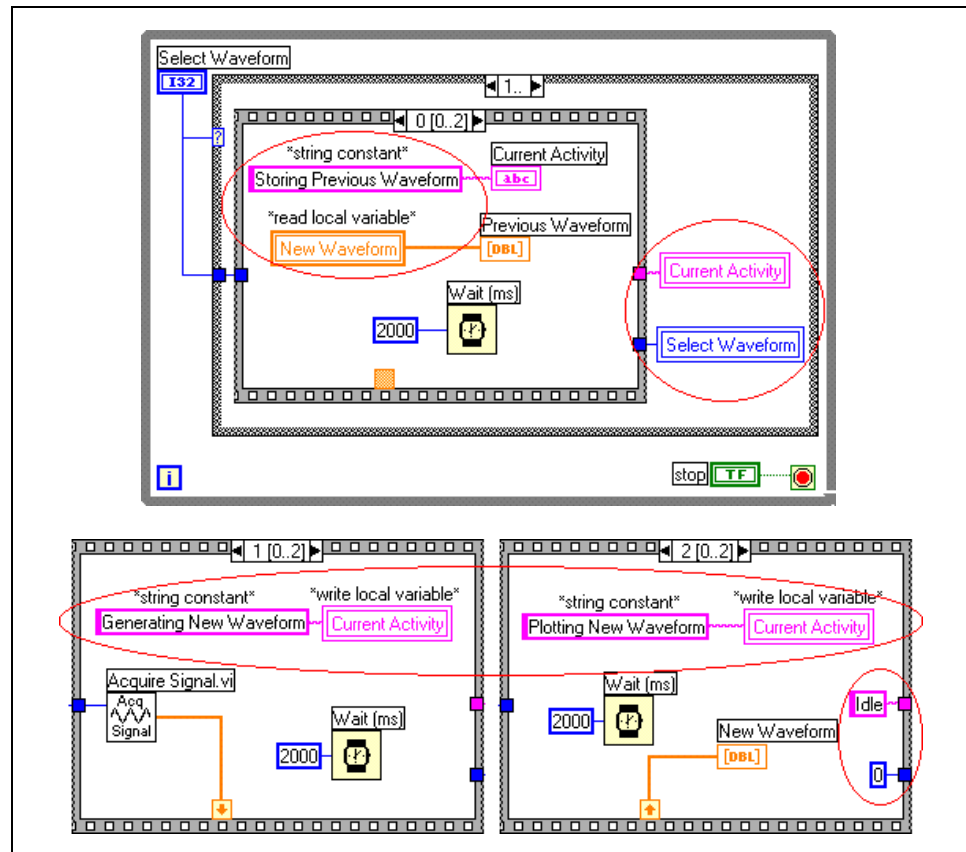
Objective: To use local variables to modify both indicators and controls on the front panel.

Front Panel



1. Open the Select and Plot Waveform VI in the `c:\exercises\Basics 2` directory. The front panel is already built. Finish the block diagram.
2. Open and examine the block diagram.

Block Diagram



1. Complete the portions of the block diagram shown in ellipses above.

string constant

a. **String Constant**, available on the **Functions»String** palette—Sends text values to the Current Activity front panel indicator.

123

b. **Numeric Constant**, available on the **Functions»Numeric** palette—Resets the Select Waveform dialog ring to zero (Stand By) after the waveform is acquired.

New Waveform

c. **New Waveform** local variable set to read local—Reads the data in the New Waveform graph and passes that data to the Previous Waveform graph. To create this local variable, right-click the **New Waveform** terminal and select **Create»Local Variable** from the pop-up menu. Then right-click the local variable and select **Change To Read**.

Current Activity

d. **Current Activity** local variable set to write local—Places new text into the Current Activity indicator. There are several instances of this local variable in the block diagram, illustrating how you can access a front panel object from several locations on the block diagram. To create the local variable, right-click the **Current Activity** terminal

and select **Create»Local Variable** from the pop-up menu. Then make two more copies of this local variable.



- e. **Select Waveform** local variable set to write local—Resets the Select Waveform control to Stand By mode. To create this local variable, right-click the **Select Waveform** terminal and select **Create»Local Variable** from the pop-up menu.



Note Make sure to correctly set each local variable as a read local or a write local. To change a write local to a read local, right-click the icon and select **Change to Read Local** from the pop-up menu. To change a read local to a write local, right-click the icon and select **Change To Write**.



- f. **Wait (ms)** function, available on the **Functions»Time & Dialog** palette—Generates a delay so you can observe the front panel activities.



- g. **Acquire Signal VI**, available on the **User Libraries»Basics 2 Course** palette—Generates the data for the waveform specified by the Select Waveform control.

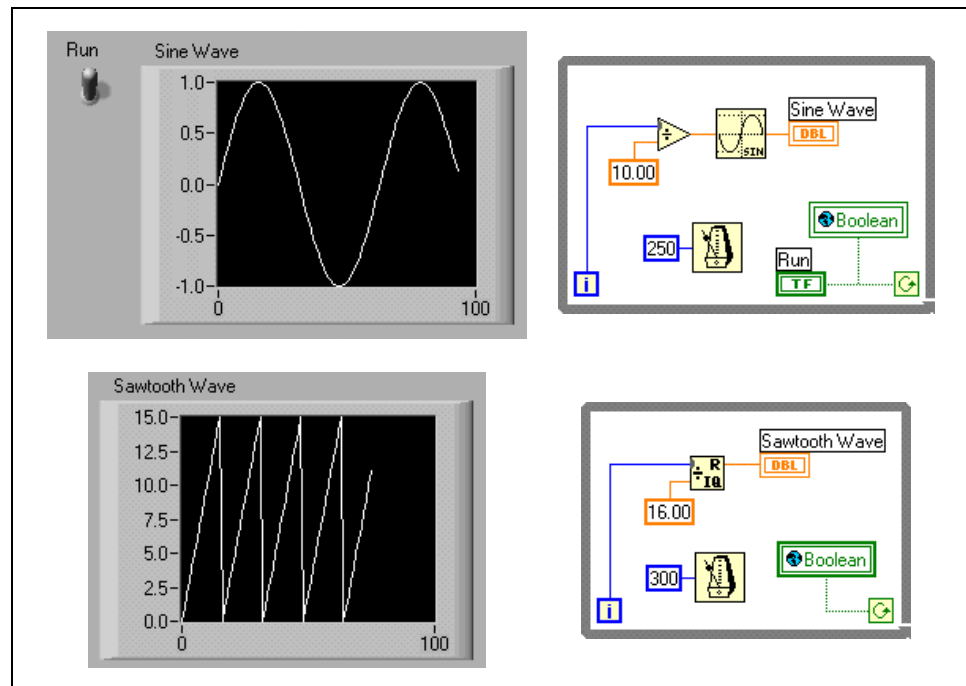
2. Save the VI under the same name.
3. Return to the front panel and run the VI. Notice that the VI updates the string indicator and stores the old waveform in the Previous Waveform graph. Also, observe that after the new waveform is acquired and sent to the screen, the Select Waveform control is reset to Stand By (zero).
4. Close the VI when you are finished.

End of Exercise 3-2

C. Global Variables

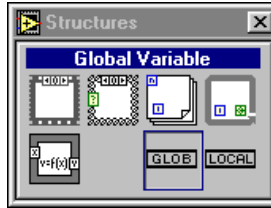
Recall that you can use local variables to access front panel objects at various locations in the block diagram. Those local variables are accessible only in that single VI. Suppose that you need to pass data between several VIs that run concurrently, or whose subVI icons cannot be connected by wires in the block diagram. You can use a global variable to accomplish this. Global variables are similar to local variables, but instead of being limited to use in a single VI, global variables can pass values between several VIs.

Consider the following example. Suppose you have two VIs running simultaneously. Each VI writes a data point to a waveform chart. The first VI also contains a Boolean to terminate both VIs. Remember that when both loops were on a single block diagram, you needed to use a local variable to terminate the loops. Now that each loop is in a separate VI, you must use a global variable to terminate the loops.



Creating Global Variables

Global variables are built-in LabVIEW objects. They appear as special VIs in the computer's memory. A global variable has a front panel where you can place controls and indicators of any type. However, a global variable has no block diagram. The front panel is simply a container from which you access data from other VIs.



Global variable node

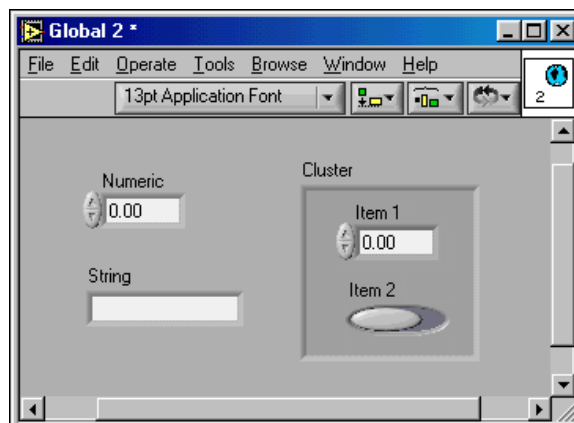
To create a global variable, select **Global Variable** from the **Structures** palette. A global variable node appears on the block diagram. The icon for a global variable on the block diagram is similar to a local variable icon, except that a small picture of a globe appears to the left of the global variable name.

Open the panel of a global variable by double-clicking the global variable node. Place controls and indicators on the front panel in the same way you place them on a standard VI's front panel.



Note You must label each control or indicator with an owned label because a global variable refers to an item by its name.

The following example shows a global variable front panel with three objects—a numeric, a string, and a cluster containing a digital and a Boolean control. Notice that the toolbar in the window does not show the same items as a normal front panel.



After you finish placing objects on the global variable's front panel, save the global variable and return to the block diagram of the original VI. Select the specific object in the global variable VI that you want to access. To select a specific object, right-click the global variable node and select the object from the **Select Item** menu. This menu lists the owned labels for all the objects on the panel. Notice that you also can open the front panel containing the objects in the global variable from this pop-up menu.

You also can click the node with the Operating tool and select the global variable you want to access.



Global variable node

After you select the specific global variable object that you want to access, the node changes from the figure shown at left to display the object you have chosen, such as a numeric.



Global variable node displaying a numeric

You might want to use this global variable in other VIs. Because a global variable is a special kind of VI, you can place it in other VIs using the **Select a VI** option in the **Functions** palette. Then, right-click the node to select the specific object in the global variable you want to access, as described above.



Note A global variable front panel can contain references to many individual objects that are globally accessible. You do not need to create a separate global variable VI for each item you need to globally access.

Read Globals and Write Globals

Like local variables, you can either read data from or write data to a global variable. By default, a global variable is *write global* when you create it. You can write a new value into the global variable, so a write global acts like an indicator.

You can change the configuration of a global variable so that it acts as a data source, or a *read global*. To do this, right-click the variable and select **Change To Read**. On the block diagram, a read global icon behaves like a control. When this node executes on the block diagram, the program reads the data in the associated front panel object.

To change a read global to a write global, right-click the variable and select **Change To Write**. The global variable will change its data direction so that it receives data instead of providing data. When this node executes on the block diagram, the program sends new data into the global variable.

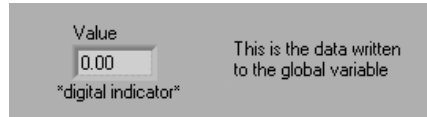
On the block diagram, you can visually distinguish read globals from write globals just as you distinguish controls from indicators. A read global has a thick border, emphasizing that it is a data source. A write global has a thin border, because it acts as a data sink. In the figure below, both global variables refer to the same item on the global variable's front panel.



Exercise 3-3 Data to Global VI

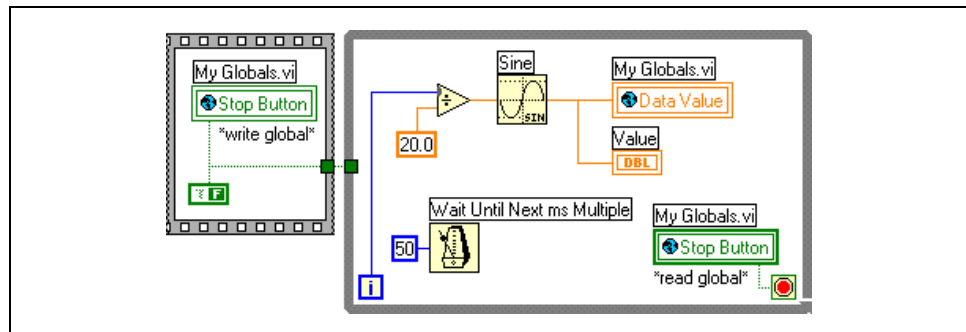
Objective: To build a VI that writes data into a global variable.

Create a global variable and use it to send data to the VI in the next exercise.

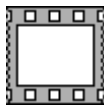


1. Open a new VI and build the front panel as shown previously.

Block Diagram



2. Build the block diagram as shown previously.



- Sequence Structure**, available on the **Functions»Structures** palette—Initializes a global variable using artificial data dependency with the Boolean Constant, available on the **Functions»Boolean** palette, and the While Loop.
- While Loop**, available on the **Functions»Structures** palette—Structures the VI to continue running until a global Boolean sends a TRUE value. Right-click the **Conditional** terminal and select **Stop If True**.
- Wait Until Next ms Multiple** function, available on the **Functions»Time & Dialog** palette—Ensures that data is written to the global variable every 50 ms, in this exercise. Create the constant by right-clicking the **input** terminal and selecting **Create»Constant**.
- Divide** function, available on the **Functions»Numeric** palette—Divides the iteration counter of the While Loop by 20. Create the constant by right-clicking the bottom **input** terminal and selecting **Create»Constant**.
- Sine** function, available on the **Functions»Numeric»Trigonometric** palette—Accepts an input value in radians and outputs the sine of that value.



Global variable node

3. **My Globals VI** global variables—Pass values between two concurrently running VIs, in this exercise. Complete the following steps to create and configure the global variables.
 - a. In the block diagram, select **Global Variable** from the **Structures** palette.
 - b. Double-click the node to bring up the global variable’s front panel. Create the global variable front panel as shown in the following figure. Label the controls with the owned labels shown. Notice that there is no block diagram associated with the global variable’s front panel.

Global Variable Panel



Global variable node

- c. Save and close the global variable. Name it `My Globals.vi`.
 - d. Return to the Data to Global block diagram.
 - e. Right-click the global variable node and select **Visible Items» Label**.
 - f. Using the Operating tool, click the global variable node and select **Stop Button**. You can change the variable to be readable or writable by right-clicking it.
 - g. Make enough copies of the `My Global.vi` variable. Use the Operating tool to select the correct item you need from the global variable.
- Initialize the Stop Button global variable inside the Sequence structure by writing a FALSE to it. The constant is wired 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.
4. Save the VI as `Data to Global.vi`. Keep it open so you can run it in the next exercise.

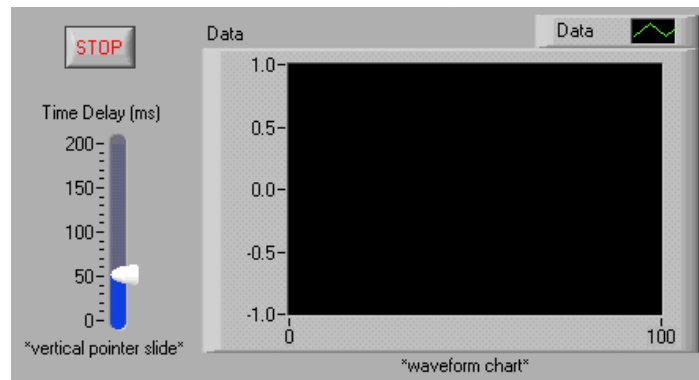
End of Exercise 3-3

Exercise 3-4 Display Global Data VI

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

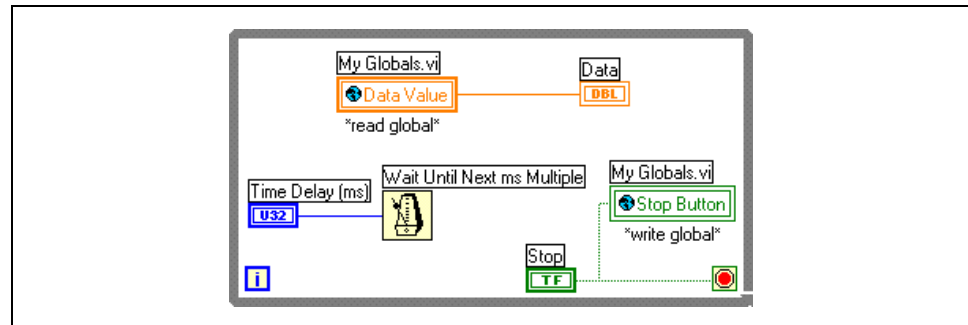
Build a VI that reads data from the global variable you created in the previous exercise and displays the data on a front panel chart.

Front Panel



1. Open a new VI and build the front panel shown previously.

Block Diagram



2. Build the block diagram shown previously.



- a. **While Loop**, available on the **Functions»Structures** palette—Structures the VI to continue running until you press the Stop button. Right-click the **Conditional** terminal and select **Stop If True**.



- b. **Wait Until Next ms Multiple** function, available on the **Functions»Time & Dialog** palette—Sets the loop rate. Make sure the default is 20 iterations per second, or 50 msec.



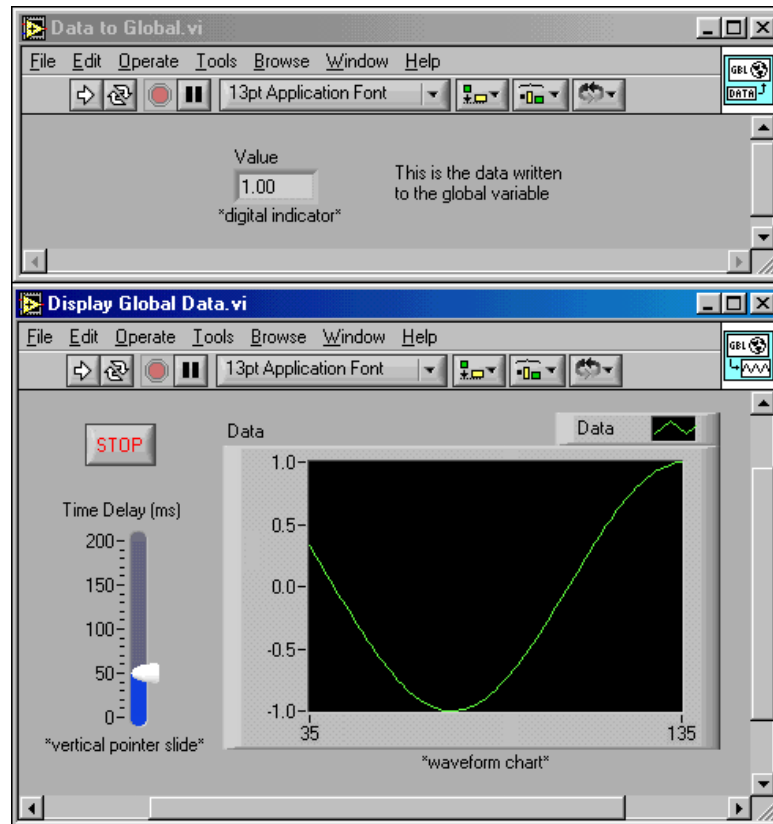
Global variable node

3. **My Globals VI** global variables—Pass values between two concurrently running VIs, in this exercise. These global variables were not created from a global variable node on this block diagram, so the steps for creating and configuring them on this block diagram are as follows.

- a. Select **Functions»Select a VI** and select `My Globals.vi` from `c:\exercises\Basics 2` directory, where you saved the global variable you created in Exercise 3-3. The global variable object displayed in the node is either **Stop Button** or **Data Value**, depending on the order in which they were placed on the global variable's front panel.
- 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.
- c. Change the Data Value global to a read global by right-clicking the node and selecting **Change To Read** from the pop-up 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. Save the VI as `Display Global Data.vi`.
5. Switch to the Data to Global VI front panel and position the two front panels so both are visible.
6. Run Data to Global. 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. Data to Global continually writes the value it calculates to the Data Value global variable object. The following example shows the Display Global Data 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 Data to Global. If you set the Time Delay to a value greater than 50 ms, Display Global Data may never read some values generated in Data to Global, 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 Data to Global, Display Global Data 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. Press the STOP button on the Display Global Data front panel to stop the VI. Notice that both Display Global Data and Data to Global stop. The VI continually writes the value of the STOP button to the Stop Button global variable object. That value is then read in Data to Global

and passed to the conditional terminal to control its loop as well. When you press the STOP button, a TRUE passes through the global variable to Data to Global, where that TRUE value is read to stop that VI as well.

8. Close both VIs.

End of Exercise 3-4

D. Important Advice about Local and Global Variables

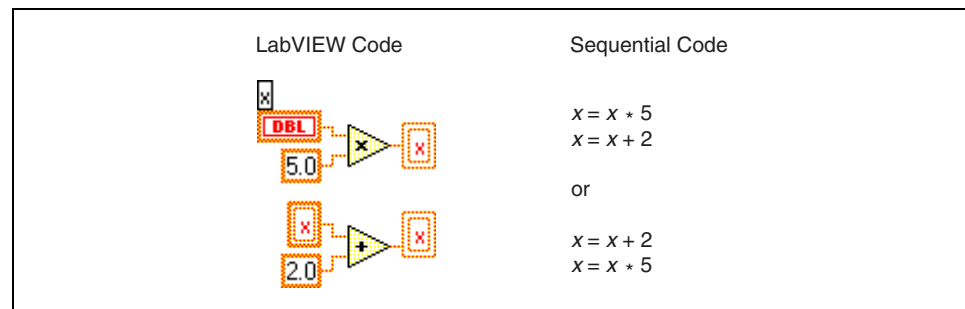
Initialize Local and Global Variables

Verify that your local and global variables contain known data values before your program begins. Otherwise, the variables may contain data that causes the program to behave incorrectly.

If you do not initialize a local variable before reading data from it, it will contain the current value of the control. The first time your program reads a global variable, it contains the default value of the object it reads, unless you have previously initialized the global variable.

Race Conditions

Local and global variables in LabVIEW do *not* behave like local and global variables in text-based programming languages. Recall that LabVIEW is a dataflow language, not a sequential programming language. Overusing local and global variables can lead to unexpected behavior in your program, because functions might not execute in the order you expect. Consider the simple local variable example below. The LabVIEW code appears next to the equivalent code of a sequential programming language.



When you execute the sequential code, the solution for a given value of x is clear because the statements execute from top to bottom. However, the LabVIEW code does not follow such a convention because LabVIEW is a dataflow language. Each node executes when all of its data is available. There are no data dependencies to guarantee the desired order of execution in the above block diagram. In fact, there are several possible solutions, depending on how the VI compiles. You cannot assume that the code located at the bottom of the block diagram executed after the code above it.

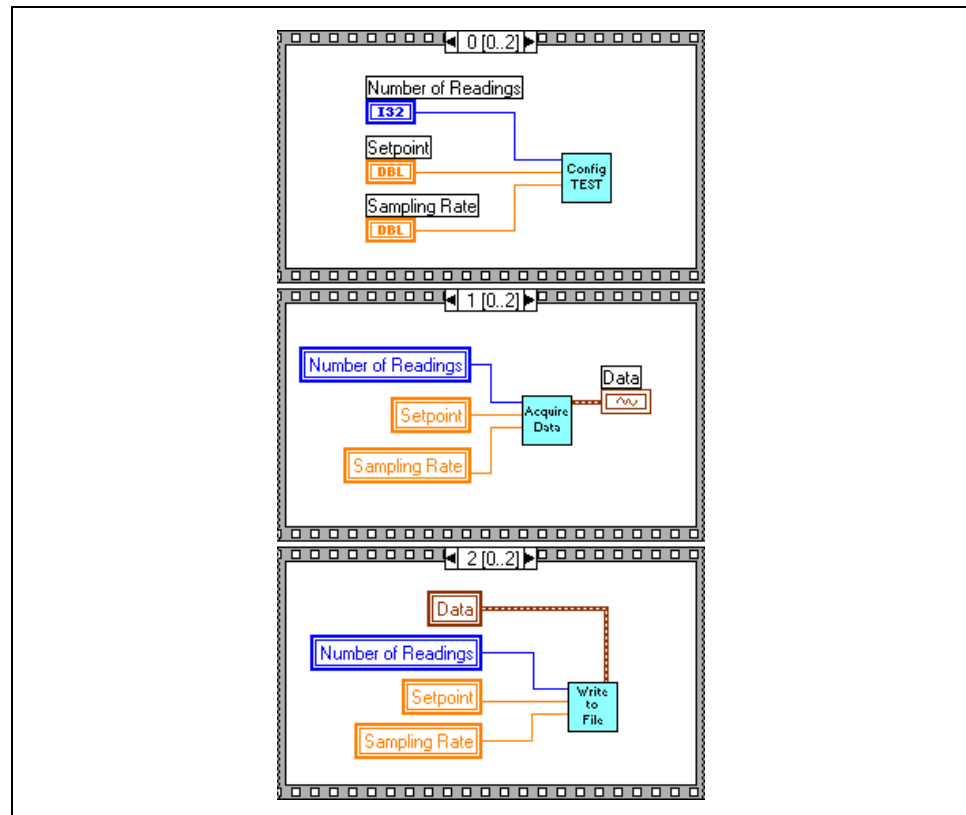
The above example illustrates what is known as a *race condition*. The result of your program depends on the order in which its instructions execute, and the code might not execute in the order you assume. If you use local and global variables, you might have a race condition if you notice that your code executes correctly only some times, or that it executes correctly during execution highlighting but not during normal execution.

To avoid race conditions, write to a local or global variable at a location separate from where you read from it—in different locations of the block diagram or structure, or in different structures or VIs. When using global variables in VIs that execute in parallel, you might want to use an additional Boolean global variable that indicates when a global variable's data has changed. Other VIs can monitor this Boolean to see if the data has changed and read the new value.

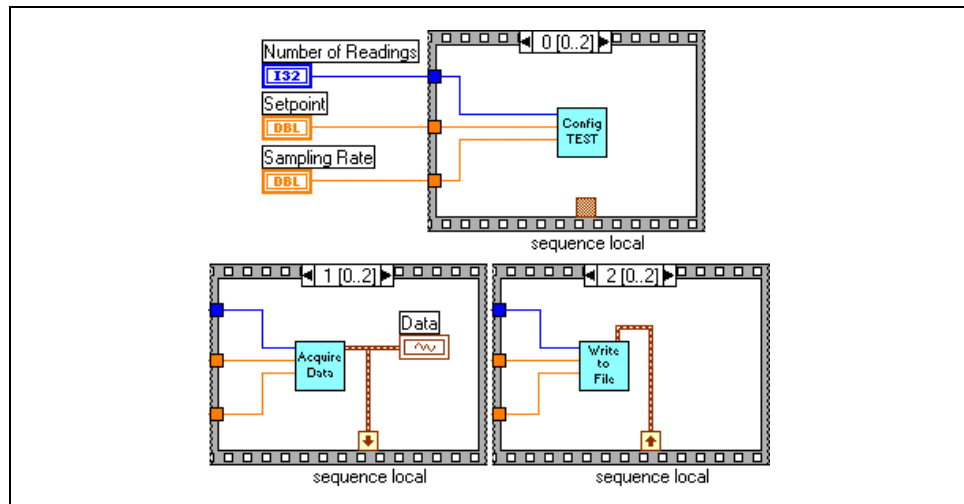
Use Local and Global Variables Only When Necessary

Local and global variables are powerful tools that you can use to accomplish useful goals. For example, you can create parallel loops on a single block diagram that are controlled with one front panel object, or you can send data between separately running VIs. However, local and global variables are inherently *not* part of the LabVIEW dataflow programming concept. Block diagrams can become more difficult to read when using local and global variables. Race conditions can cause unpredictable behavior. Accessing data stored in a local or global variable is slower than using dataflow and less memory efficient. Therefore, use local and global variables sparingly and only when necessary.

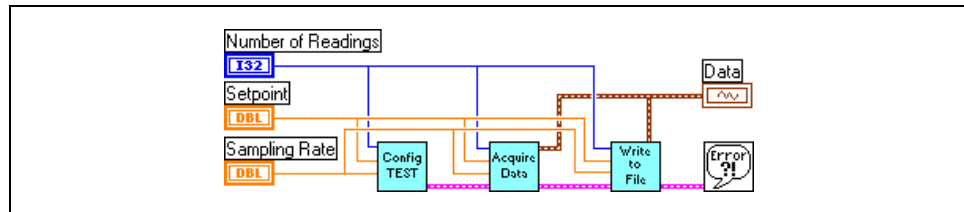
For example, do not use locals and globals to avoid long wires across the block diagram or to access values in each frame of a sequence structure. Consider the following block diagram.



This three-frame 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 there is nothing inherently wrong with using the sequence structure and the local variables in this way, 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 block diagram shown above eliminates the use of local variables by putting the original control terminals outside the 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 block diagram shown above removes the sequence structure and uses dataflow 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.

E. DataSocket

So far in this lesson you learned three main things:

- The most efficient way to manage data in the block diagram is through a direct wire connection.
- Local variables can be used to access front panel objects in multiple places in the block diagram.
- Global variables can be used to pass data between separate VIs without using a wire.

There are many other ways to manage data in LabVIEW. This lesson covers one other method, DataSocket. DataSocket is an Internet programming technology based on TCP/IP that simplifies data exchange between computers and applications. With DataSocket, you can efficiently pass data over the Internet and respond to multiple users without the complexity of low-level TCP programming.

Therefore, you can use DataSocket to pass live data not only between VIs running on the same machine but also between VIs running on separate computers that are connected through a network. You also can use DataSocket to communicate between LabVIEW and any other programming language that contains support for TCP/IP, such as Excel, Visual Basic, C, and so on.

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

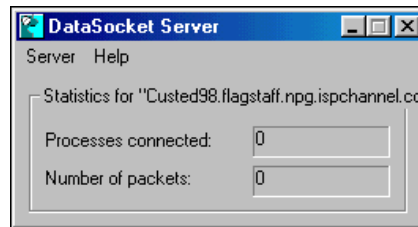
DataSocket is a single, unified, end-user API based on URLs for connecting to measurement and automation data located anywhere, be it on a local computer or on the Internet. It is a protocol-independent, language-independent, and OS-independent API designed to simplify binary data publishing. The DataSocket 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.



As shown above, 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 **Hide DataSocket Server** from the Server menu.

Broadcasting data with the DataSocket Server requires three actors—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 actors can reside on the same machine, but more often the three actors 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

Before you can start using the DataSocket functions in LabVIEW, you need to understand how DataSocket connects to different I/O technologies and how you name the device or resource you are transferring data to or from. For example, in file I/O the resource name is a file path. For TCP/IP there are two parts to the name, a machine name and a port number. With DataSocket, the resource name is in the form of a URL, or uniform resource locator, much like the familiar web address used by a web browser. Consider how a web browser would interpret the URL `ni.com/datasocket`. It tells the browser to use the TCP/IP-based protocol called HTTP, or hyper text

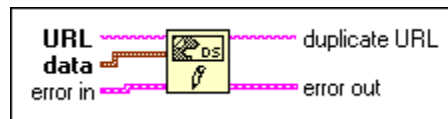
transfer protocol, to connect to the machine named ni.com and to fetch the web page named datasocket.

The URL is different from the names used by most I/O technologies in that it not only defines what you are interested in but also indicates how to get it. The how encoded in the first part of the URL is called the access method or 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 in the front tells DataSocket to open a data socket transfer protocol connection to the test machine and fetch a signal called wave1. Had the URL started with file, the data would have been fetched from a file instead of the DataSocket server.

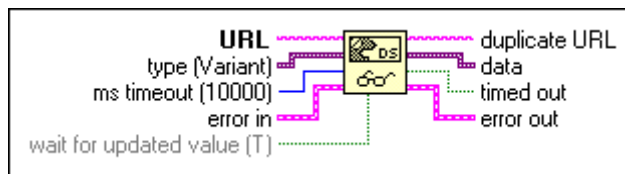
The DataSocket Functions

The DataSocket API in LabVIEW is a palette of functions and VIs located in the **Functions»Communication»DataSocket** palette.

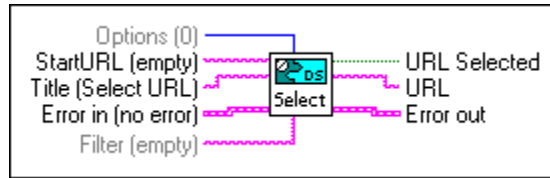
DataSocket communication is simple. You can do all the basic operations with the following functions and VIs.



The DataSocket Write function writes the data input to the specified URL. The data can be in any format or LabVIEW data type. The **error in** and **error out** clusters maintain the error conditions.



The DataSocket Read function reads the data type specified by type from the specified URL. The **ms timeout** value has a default of 10 seconds and you can specify a different timeout value. The **timed out** Boolean indicates a TRUE if this function timed out. The **error in** and **error out** clusters maintain the error conditions.



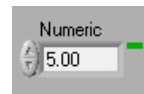
The DataSocket Select URL VI is only used when you do not know the URL for an object and you want to search for a data source or target from a dialog box.

Direct DataSocket Connection to Any Panel Object

You do not need to use the DataSocket Write and DataSocket Read functions to publish and subscribe data using the DataSocket Server. You can establish a DataSocket connection from a front panel object. Right-click any front panel object and select **Data Operations»DataSocket Connection** to display the **DataSocket Connection** dialog box.

In the **Connect To** field, enter a URL and select whether you want to publish, subscribe, or both. Then check the **Enabled** box.

When you click the **Attach** button, the front panel object is now available at the specified URL in the DataSocket Server. A small rectangle in the top right of the control or indicator appears, indicating the status of the DataSocket connection. If the rectangle is green as shown below, the connection is good.



If the rectangle is gray, there is no connection to the server at that time. If the rectangle is red, the connection has an error.

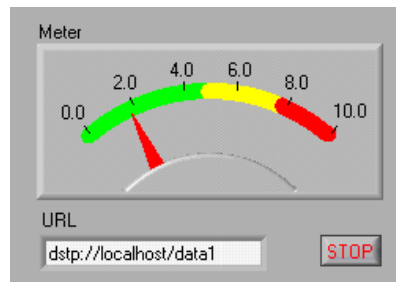
Now build two VIs that transfer data using DataSocket.

Exercise 3-5 DS Generate Data VI and DS Read Data VI

Objective: To build two VIs that use DataSocket to transfer data.

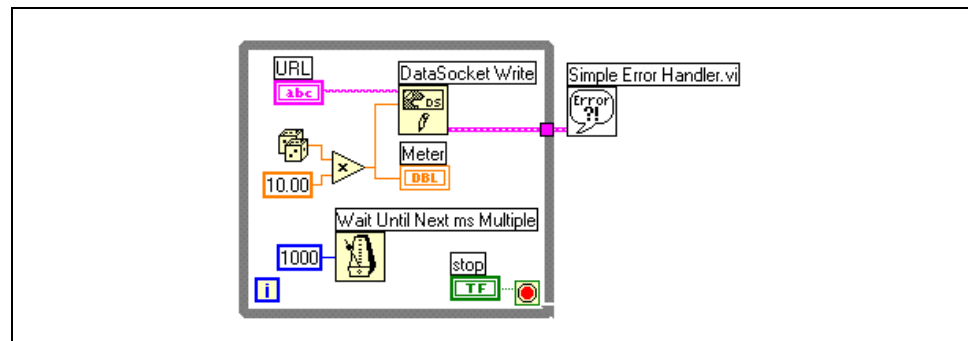
Build a VI that generates a random number and displays this value in a meter and sends the data to a DataSocket URL. Then build a second VI that reads the DataSocket URL and displays the value in a slide. Last, use the automatic publish and subscribe features to send the data through a DataSocket connection.

Front Panel







1. Open a new VI and build the front panel shown above. The URL object is a string control.

Block Diagram

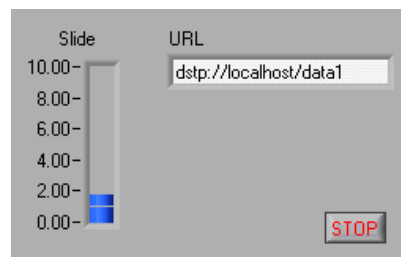


2. Open and build the block diagram shown above using the following components:
 - a. **While Loop**, available on the **Functions»Structures** palette—Structures the VI to continue running until you press the Stop button. Right-click the **CONDITIONAL** terminal and select **Stop If True**.
 - b. **Wait Until Next ms Multiple** function, available on the **Functions»Time & Dialog** palette—Causes the While Loop to execute once per second. Create the constant by right-clicking the **input** terminal and selecting **Create»Constant**.



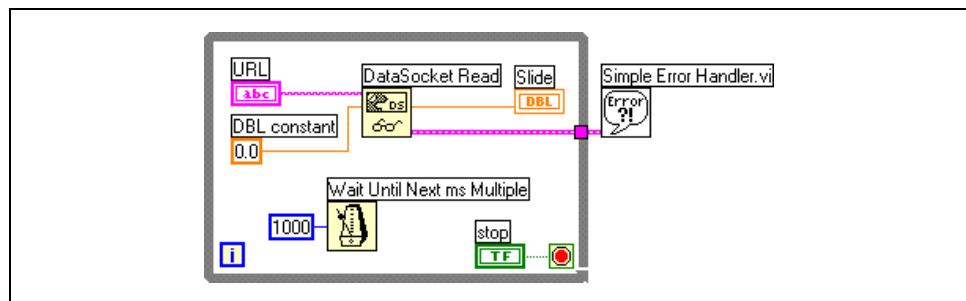
-  c. **Random Number (0-1)** function, available on the **Functions»Numeric** palette—Creates a random number between zero and one.
 -  d. **Multiply** function, available on the **Functions»Numeric** palette—Multiplies two numbers together and is used here to scale the random number to be between zero and 10.
 -  e. **DataSocket Write** function, available on the **Functions»Communication»DataSocket** palette—Writes the random data value to the specified URL.
 -  f. **Simple Error Handler VI**, available on the **Functions»Time & Dialog** palette—Opens a dialog box if an error occurs and displays the error information.
3. Save this VI as `DS Generate Data.vi`.
 4. Now build the second VI to read the random value.

Front Panel



1. Open a new VI and build the front panel shown above. The URL object is a string control. To show the scale on the slide indicator, right-click the slide and select **Scale»Style**.

Block Diagram



2. Open and build the block diagram shown above using the following components:
 - a. **While Loop**, available on the **Functions»Structures** palette—Structures the VI to continue running until you press the



Stop button. Right-click the **Conditional** terminal and select **Stop If True**.



- b. **Wait Until Next ms Multiple** function, available on the **Functions»Time & Dialog** palette—Causes the While Loop to execute once per second. Create the constant by right-clicking the **input** terminal and selecting **Create»Constant**.



- c. **Numeric Constant**, available on the **Functions»Numeric** palette—Creates the correct data type to read the value through DataSocket. Make sure this constant is a DBL by right-clicking it and selecting **Representation»Double Precision** from the shortcut menu.

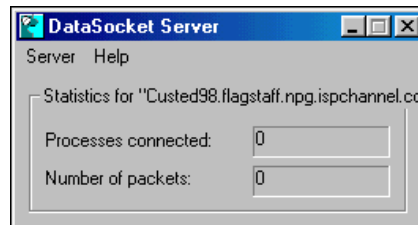


- d. **DataSocket Read** function, available on the **Functions»Communication»DataSocket** palette—Reads the random data value from the specified URL.



- e. **Simple Error Handler VI**, available on the **Functions»Time & Dialog** palette—Opens a dialog box if an error occurs and displays the error information.

3. Save this VI as `DS Read Data.vi`.
4. Position the front panels of the DS Generate Data and DS Read Data VIs so that you can see both front panels.
5. Start the DataSocket Server by going to the Start menu and selecting **Programs»National Instruments»DataSocket»DataSocket Server**. The **DataSocket Server** window, similar to the one below, appears.

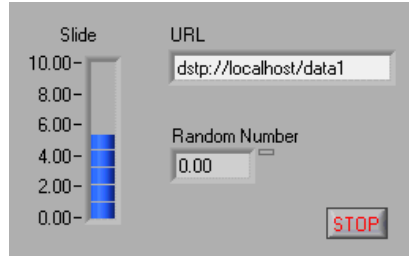


6. Return to the two VI front panels and make sure that the URLs have been entered the same for both VIs.
 - **dstp**—The DataSocket transfer protocol.
 - **localhost**—The current computer you are using.
 - **data1**—The name given to the random number you will be sending.
7. Run the DS Generate Data and the DS Read Data VIs.

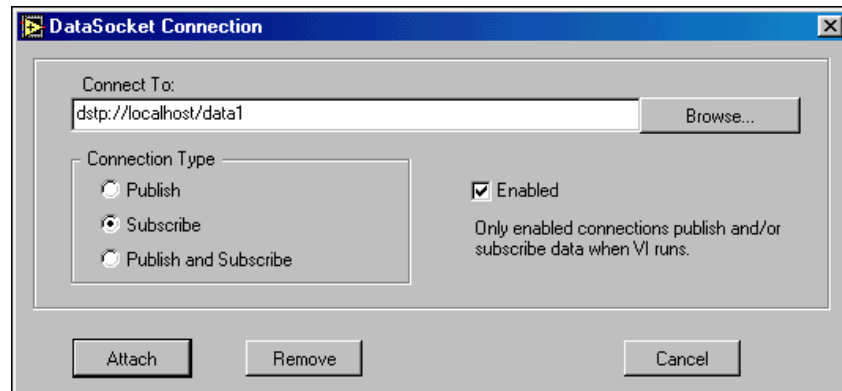
The **DataSocket Server** window shows one process connected, and the Number of packets value increments each second as the meter and slide show the same random numbers.

- Stop both VIs when you are finished. Now modify the VIs to use the automatic publish and subscribe capabilities of front panel objects.

Front Panel



- Place a digital indicator on the front panel of the DS Read Data VI as shown above.
- Right-click the new digital indicator and select **Data Operations» DataSocket Connection** from the shortcut menu. Enter the following values.



- Click the **Attach** button and a little gray rectangle appears to the top right side of the digital indicator. This indicates that the DataSocket connection is not active.
- 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 type in 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. Go to the National Instruments DataSocket web page, at ni.com/datasocket, for more information.

13. Stop and close both VIs and the DataSocket Server when you are finished.

End of Exercise 3-5

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»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 use global and local variables to access a given set of values throughout your LabVIEW application. These variables pass information between 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 between 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 cause slower execution and inefficient memory usage in your application.
- Local and global variables do not use dataflow, so if you use them too frequently, they can make your block diagrams difficult for others to understand. Use locals and globals wisely.

Additional Exercises

3-6 Open the In Range VI. 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, available on the **Functions»Comparison** palette. Modify the VI using a local variable so that the LED is turned off when the VI starts execution. Save your VI after you have finished.

3-7 As mentioned earlier, 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 3-4, where you could adjust the time delay between reads of the Data Value global faster or slower than the data was actually available.

Modify Data to Global and Display Global Data, from Exercises 3-3 and 3-4, 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, which will be used to indicate when VIs are ready to send or receive data. In the Data to Global VI, make sure to set the Handshake global Boolean to FALSE before it passes a new data point to the Data Value numeric. In Display Global Data, set the Handshake Boolean to TRUE before it attempts to read the numeric data. In both VIs, make sure to set the Handshake Boolean so the other VI knows when to access the variable.

Name the new global Boolean Handshake Global.vi, and the other two VIs Data to Handshake Global.vi and Display Handshake Global.vi.

Challenge

3-8 Open the Select and Plot Waveform VI from Exercise 3-2. Use the information at the end of Section D to rewrite this VI to no longer use the sequence structure and local variables.



Tip Use a state machine architecture and pass data using shift registers or direct wire connections.

Save the VI as Select and Plot Waveform2.vi.

Notes

Notes

Lesson 4

Advanced File I/O Techniques



When implementing subVIs for file I/O, you might need to account for different file formats. For example, if a third-party application needs to read data acquired by LabVIEW, you should save that data in text format because most applications have support for text files. On the other hand, if only LabVIEW will access the data, and file size is critical, binary files are a better choice. Advanced File I/O techniques and the built-in file I/O functions allow this flexibility.

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.

Application SubVIs You Will Build:

- A. Save Data to File VI
- B. View Analysis File VI

A. Working with Byte Stream Files

LabVIEW's file I/O functions make working with byte stream files, or text and binary files, quite simple. You use the same functions to manipulate these files. However, when working with binary files, the data is generally not in a readable form. 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.

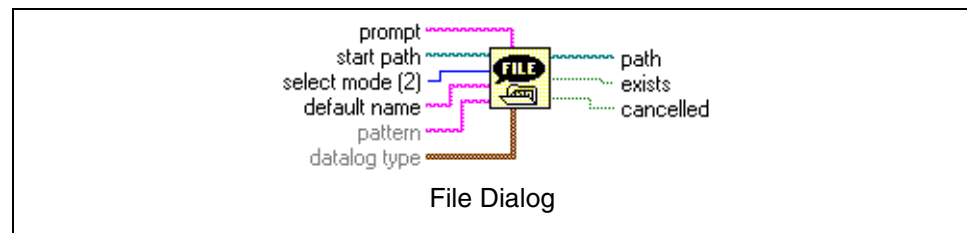
Frequently Used File I/O Functions

Before describing byte stream files in detail, we will first briefly review the basic LabVIEW file I/O functions. These functions are located on the **Functions»File I/O** and **Functions»File I/O»Advanced File Functions** palettes.

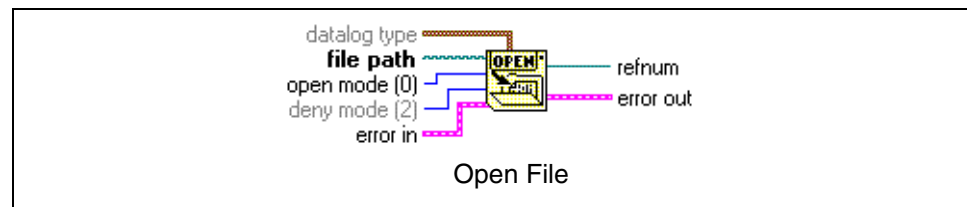


Note Refer to the **Help»Contents and Index** for more information about these functions.

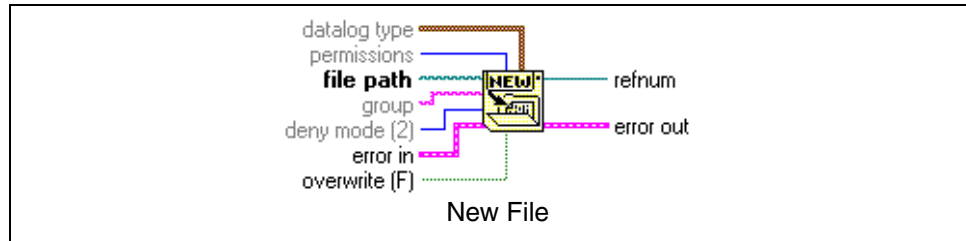
The File Dialog function on the **File I/O»Advanced File Functions** palette displays a file dialog box for file selection. You can select new or existing files or directories from this dialog box. We will describe the use of the **datalog type** input later in this lesson.



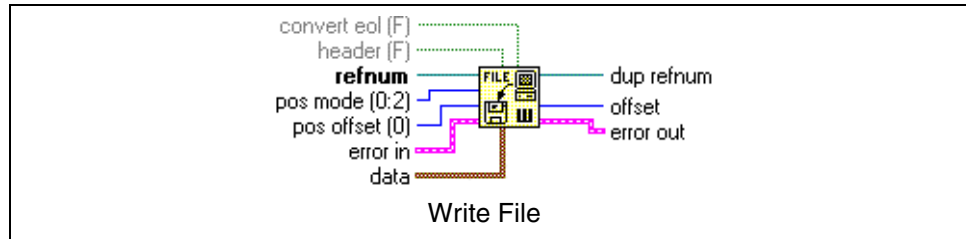
The Open File function on the **File I/O»Advanced File Functions** palette opens an existing file. You must wire a valid path to the **file path** input. This function is *not* capable of creating or replacing files. It opens only *existing* files. The **datalog type** input is used only when opening LabVIEW datalog files.



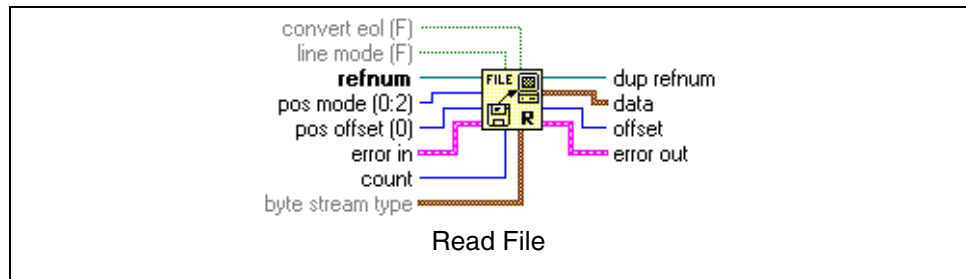
The New File function on the **Functions»File I/O»Advanced File Functions** palette creates a new file for reading or writing. You must wire a path to the file path input. Use the **datalog type** input only when creating new LabVIEW datalog files.



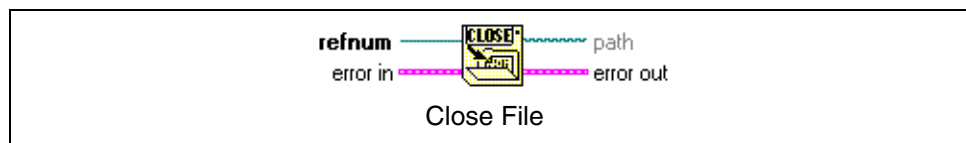
The Write File function on the **Functions»File I/O** palette writes data to an open file. The behavior of this function varies slightly depending on whether you are writing data to a byte stream file or to a LabVIEW datalog file.



The Read File function on the **Functions»File I/O** palette reads data from an open file. When reading byte stream files, you can use the **byte stream type** input to indicate how LabVIEW should interpret data in the file. We will describe this concept later in this lesson.

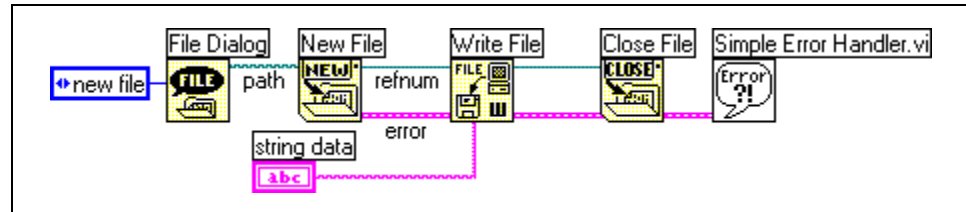


The Close File function on the **Functions»File I/O** palette closes the file associated with **refnum**.



Byte Stream Files

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.



When to Use Text Files

Text files are useful and common for many reasons. Almost every operating system and a majority of software applications can read and write files in text format. Historically, most instrument control applications, such as GPIB and serial, use text strings to send control statements.

Because of the universality of the text format, there are several situations where text files are preferable for data storage. For example, text files are preferable when you plan to read or manipulate the data with other software applications, such as spreadsheet or word processing applications.

Text files also have some significant disadvantages. It is difficult to randomly access numeric data in text files. Although each character in a string takes up exactly one 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.

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.

As shown in the previous block diagram, the Write File function skips over any header information that LabVIEW uses to store the string in memory and writes the contents of the string data control to the file. In fact, the Write File function does not distinguish a text string from a binary string when writing the data to the file—it places the data in the file. The data terminal of the Write File function is polymorphic, which means that it adapts to any kind of data you wire into it. Thus, you can create a binary file by wiring binary data to the Write File function in the previous example. However, you will find that header information is vital for interpreting the binary file.

For example, if you wire a two-dimensional 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, nor does it place

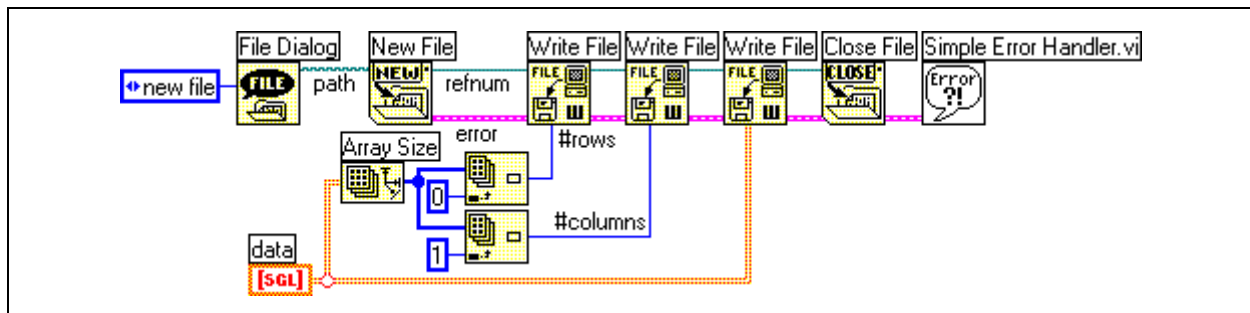
any information about the number of rows or columns in the array into the file. Even if you know that the data were originally single-precision floating-point numbers, you cannot successfully reconstruct the two-dimensional array. If the file contains 24 bytes of data, and knowing that single-precision floating-point numbers use four bytes of memory each, you can figure out that the array contained six values ($24/4 = 6$). However, the original data might have been stored in a one-dimensional array of six elements, a two-dimensional array with one row and six columns, or a table with three columns and two rows.

When to Use Binary Files

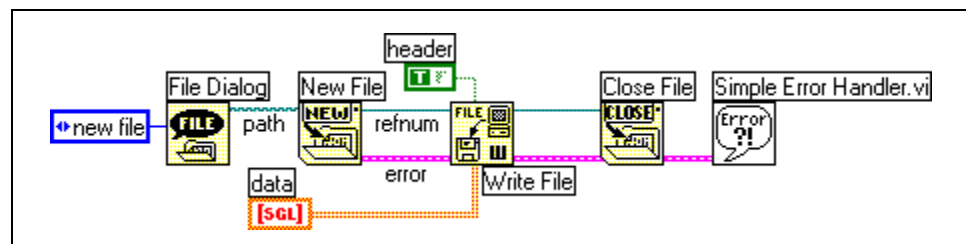
Compared to text files, binary files require less disk space to store the same amount of information. Binary files allow you to randomly access numeric data stored in the file. However, binary files are generally not portable to other applications. Also, you must carefully document the details needed to extract information stored in a binary file.

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.

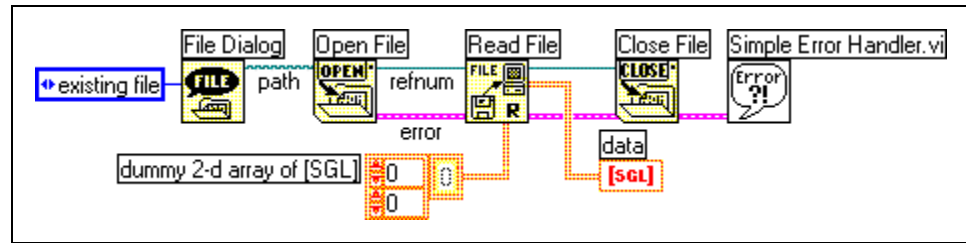


You also can generate the same file using the **header** input of the Write File function. If you wire a Boolean value of TRUE to the **header** input of the Write File function, it writes the same data to the file as if you had manually written a header. The following example shows how to use the Write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



Using the previous block diagram example, if you again wire a two-dimensional array of single-precision numbers with three rows and two columns, the file would contain 24 bytes of data and two additional long integers, four bytes each, as header information. The first four bytes contains the number of rows in the array; the second four bytes contains the number of columns.

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

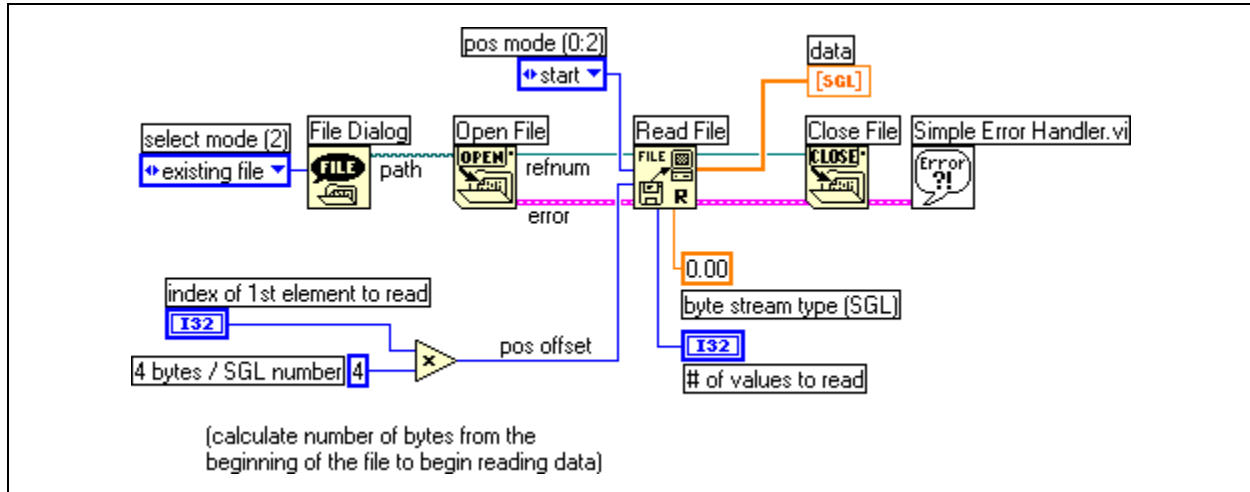


Notice that the **byte stream type** input of the Read File function has a two-dimensional array of single-precision numbers wired to it. The Read File function uses only the data type of this input to read the data in the file, assuming that it has the correct header information for that data type.

Random Access in Byte Stream Files

If you store arrays of numeric data in a file, you might find it necessary to access data at random locations in the file. You cannot randomly access data stored in text files when the data contains negative signs, varying number of digits in individual data points, and other factors. For example, if you have an array of 100 numbers ranging in value from 0 to 1,000, you cannot predict where a given element in the array is located within the text file. The problem is that in text, the number 345 requires three bytes of storage, while the number 2 requires only one byte. Therefore, you cannot predict the location of an arbitrary array element in the file.

Such obstacles do not occur in binary files. In a binary file, the flattened format of a number in LabVIEW is a binary image of the number itself. Therefore, each number in the array uses a fixed number of bytes of storage on disk. If you know that a file stores single-precision numbers, which use four bytes per number, you can read an arbitrary group of elements from the array, as shown in the following example.



In the previous block diagram 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 $4 \times n$ bytes from the beginning of the file. A single-precision floating-point constant, wired to the **byte stream type** input, tells the Read File function to read single-precision floating-point values from the file. The # values to read control, connected to the **count** input of the Read File function, tells the function how many single-precision elements to read from the file. Notice that when the **count** input is wired, LabVIEW places the output data in an array, because LabVIEW reads more than one value from the file.

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

- When performing text and 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 the **byte stream type** input is wired, the **data** output of the Read File function is of the same data type as the **byte stream type** input when the **count** input is unwired.
- If the **byte stream type** input is wired and the **count** input has data connected to it, then the Read File function returns an array containing **count** elements of the same data type as the **byte stream type** input.
- Refer to the LabVIEW Help and the *LabVIEW User Manual* for more information about random access operations and byte stream files.

Exercise 4-1 Binary File Writer VI

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

This VI saves data to a binary file using a simple formatting scheme in which the file's header is a long word integer (I32) containing the number of data points in the file. In the next exercise, build a VI that reads the binary file.

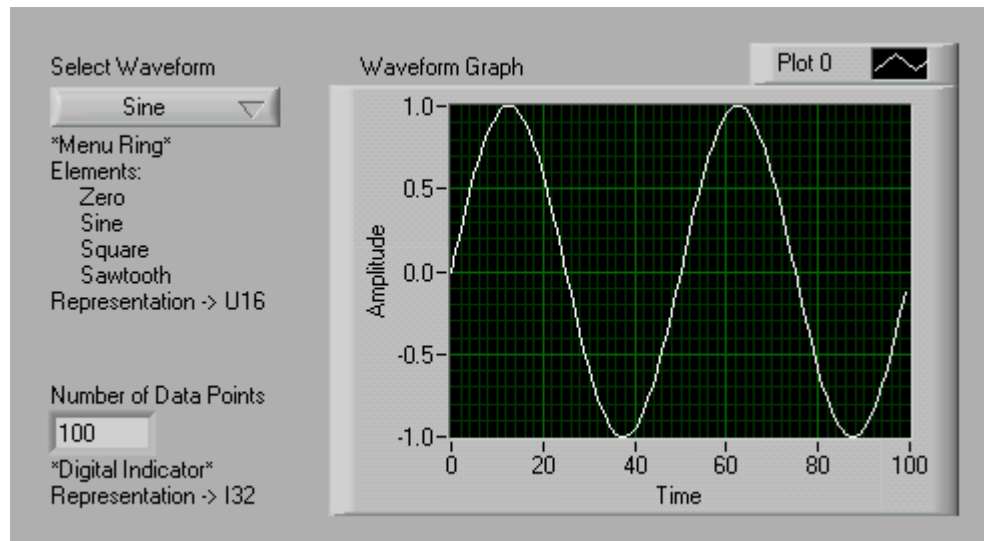
Front Panel

1. Open a new VI.
2. Build the following front panel. When you create the menu ring, recall that a shortcut for adding a new item to the list of options is to press <Shift-Enter> after entering the text for an item in the list. Enter *Zero* for item 0 in the list, *Sine* for item one, and so on. To view the numeric value of the menu ring, right-click the control and select **Visible Items»Digital Display**.

Macintosh, <Shift-Return>; Sun, <Shift-Return>; HP-UX, <Shift-Enter>; and Linux, <Shift-Enter>



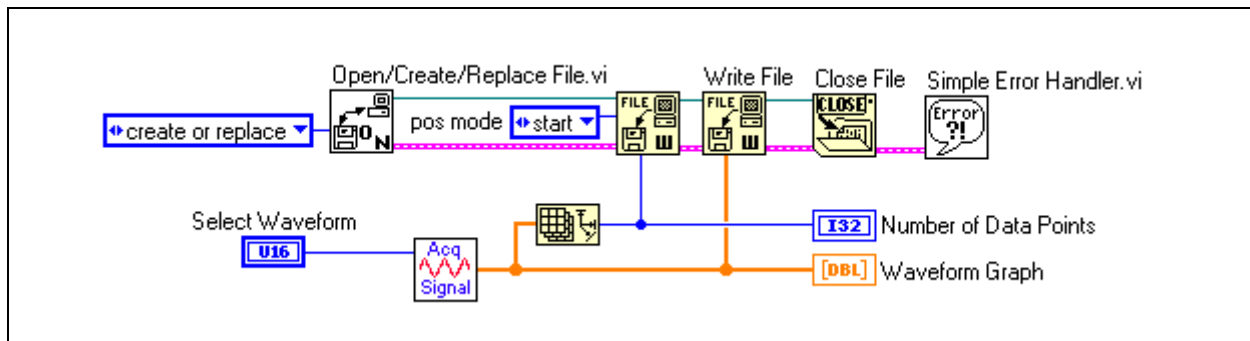
Note The appearance of the menu ring control varies from one platform to the next.



3. Open the block diagram.

Block Diagram

4. Build the following block diagram.



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



- b. Place the Write File function located on the **Functions»File I/O** palette on the block diagram. This function appears twice in this exercise. The first function 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.



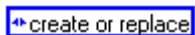
- c. Place the Close File function located on the **Functions»File I/O** palette on the block diagram. In this exercise, this function closes the binary file after data has been written to it.



- d. Place the Simple Error Handler VI located on the **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. 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.



- f. 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.



- g. Place the Acquire Signal VI located on the **Functions»User Libraries»Basics-II Course** palette on the block diagram. In this exercise, this VI generates the waveform selected by the Select Waveform control.



- h. Place the Array Size function located on the **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.
5. Save the VI as `Binary File Writer.vi`. Run it, saving data to the file on disk named `data.bin`.

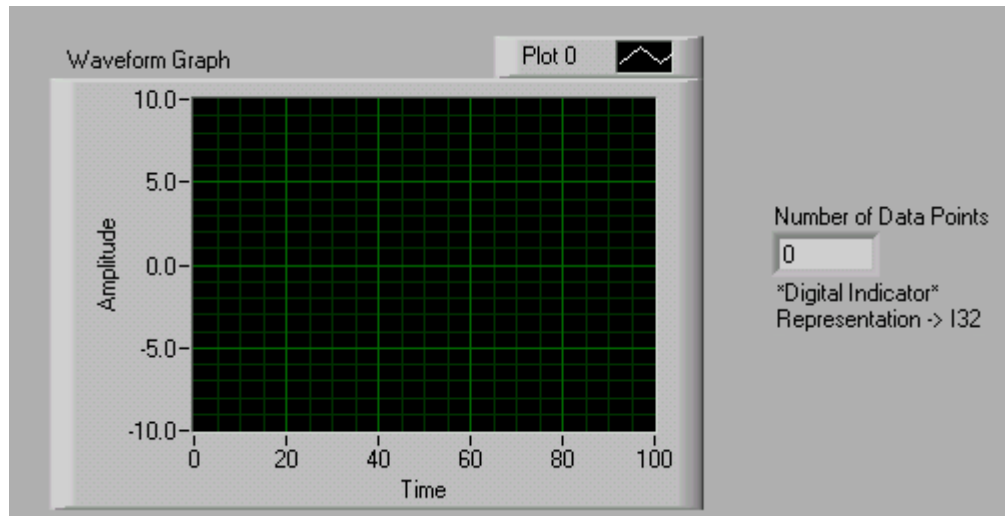
End of Exercise 4-1

Exercise 4-2 Binary File Reader VI

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

Front Panel

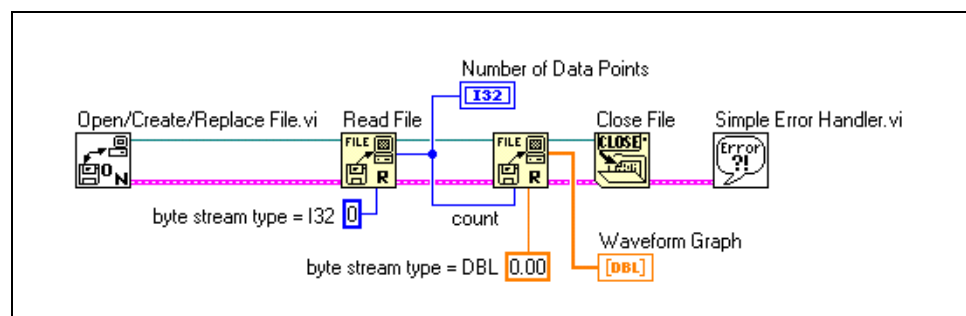
1. Open a new VI and build the following front panel.



2. Open the block diagram.

Block Diagram

3. Build the following block diagram.



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



- b. Place the Read File function located on the **Functions»File I/O** palette on the block diagram. This function appears twice in this exercise. The first function 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»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»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»Numeric** palette on the block diagram. You must create an I32 constant 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»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**.

4. Return to the front panel. Save the VI as `Binary File Reader.vi`.
5. Run the VI. Open the `data.bin` file created in Exercise 4-1.

After the file opens, the Read File function uses the **byte stream type** input, which has a long integer, four bytes, wired to it, 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 4-1.

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.

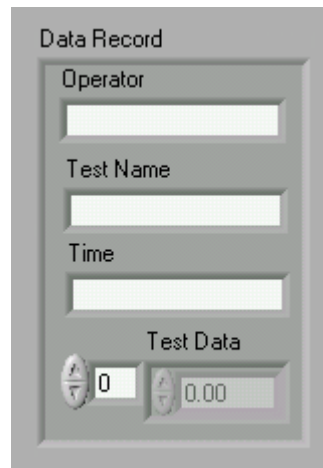
6. Close the VI.

End of Exercise 4-2

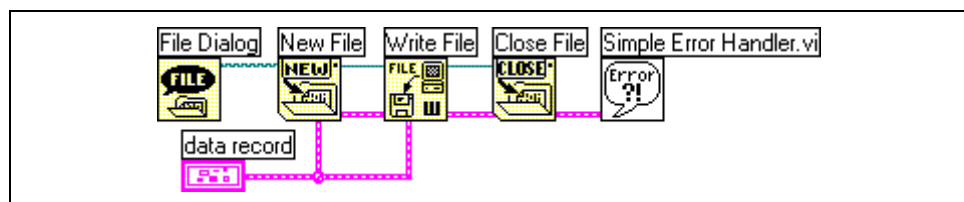
B. LabVIEW Datalog Files

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. For example, you might want to save records of data that contain several hundred data points, including a date and time stamp for each set.

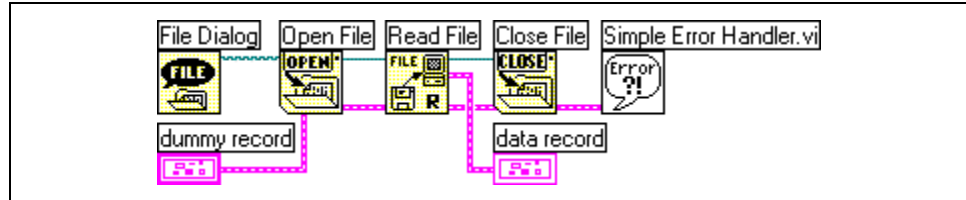
A datalog file stores information as a series of records of any data type. Although all of the records in a file must be of the same type, those records can be arbitrarily complex. Each record is written to the file as a cluster containing the data to be stored. In the following example, each record 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.



Use the same file I/O functions to work with datalog files that you use for byte stream files. However, you use the data type input differently for datalog files. 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 need not be an actual data record—you need it only to establish the type of data that you can store 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.

When to Use Datalog Files

You can use datalog files to store and retrieve complex data formats in LabVIEW. However, datalog files, like binary files, do not have an industry-standard format for storage. Thus, they are virtually impossible to read with software applications other than LabVIEW. Datalog files are most useful if you intend to access the data only from LabVIEW and need to store complex data structures.

Exercise 4-3 Save Data to File VI

Objective: To finish a VI that saves data in a text, binary, or datalog file.



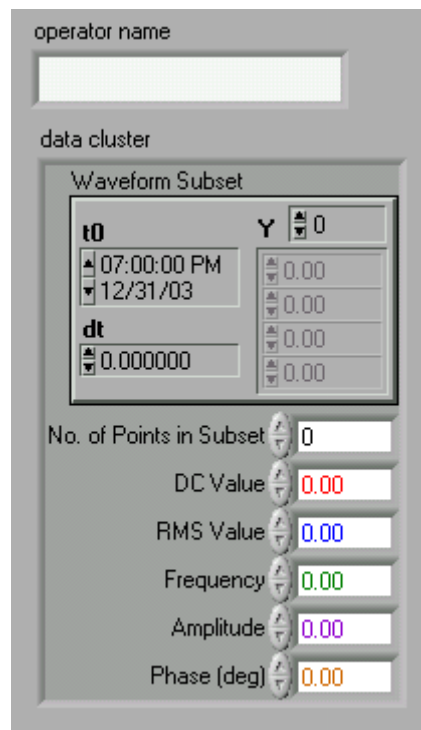
Note Completing this VI enables the Log Results to File option in the Analyze and Present Data VI from Exercise 2-7. Recall that you will use Exercise 4-3 in the project in Lesson 5.

In the application being developed, you need to save a mixed data set to disk. This data set contains simple numerics, arrays of numerics, and string data. 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 in `exercises\Basics2` directory. The front panel is already built.

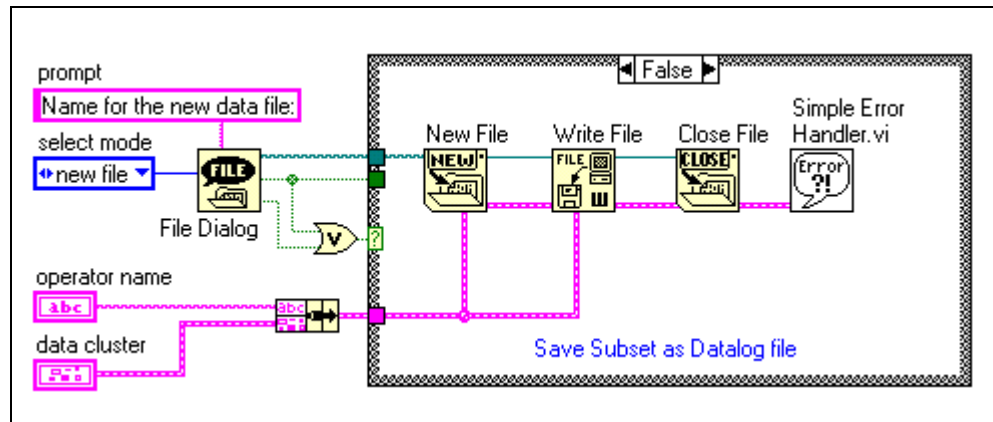
The two controls on the front panel are used to pass application data to this VI, which functions as a subVI in the finished project. The Data Cluster will contain the subset of analyzed data to save to disk, and the Employee Name string will contain the Operator name to save to the file.



2. Open the block diagram.

Block Diagram

3. Complete the Case structures for the following block diagram. The cases that are not shown are already built.



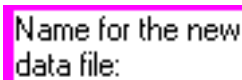
- a. Delete the Boolean constant labeled `Delete Me`.



- b. Place the File Dialog function located on the **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.



- c. To create this constant, right-click the **select mode** input of the File Dialog function and select **Create»Constant**. With the Operating tool, set this constant to the value `new file`.



- d. To create this constant, right-click the **prompt** input of the File Dialog function and select **Create»Constant**. This string displays a prompt message in the file dialog box.

The outside Case structure writes the data to a LabVIEW datalog file. Notice that bundling the Employee Name and Data Cluster together provides the data type for the datalog file.



- e. Place the New File function located on the **Functions»File I/O»Advanced File Functions** palette on the block diagram. This function creates the new file. Notice that the **datalog type** input to this function must receive an input.



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



- g. Place the Close File function located on the **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»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. After you finish the VI, save and close it.
5. Open `Analyze and Present Data.vi`, which you completed in Exercise 3-3. This VI calls `Save Data to File VI`. When you run this VI and click the **Log Results to File** button, a dialog box in the `Save Data to File VI` appears so you can name the data file to save. Once you select the filename, LabVIEW saves the data set as a datalog file.

End of Exercise 4-3

Exercise 4-4 View Analysis File VI

Objective: To study a VI that reads data files created by the Save Data to File VI from Exercise 4-3.

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



Note You will use this VI in Lesson 5.

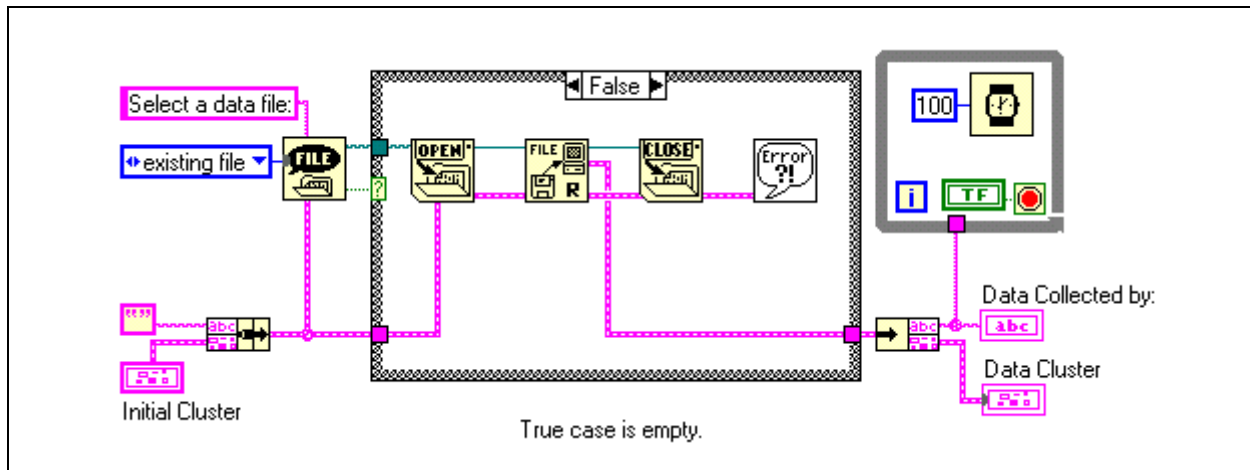
Front Panel

1. Open the View Analysis File VI in `exercises\Basics2` directory. This VI has already been built.



2. Open the block diagram.

Block Diagram



3. Examine the behavior of the VI. If the user cancels the file dialog box, nothing happens.

Notice that the **datalog type** input of the File Dialog function is wired to a dummy cluster of the same data type to be read. This connection causes the LabVIEW file dialog to display only directories and files of the appropriate data type. Once the file is selected, the file is opened as a datalog file and a single data record is read. Finally, notice the use of error checking in this application.

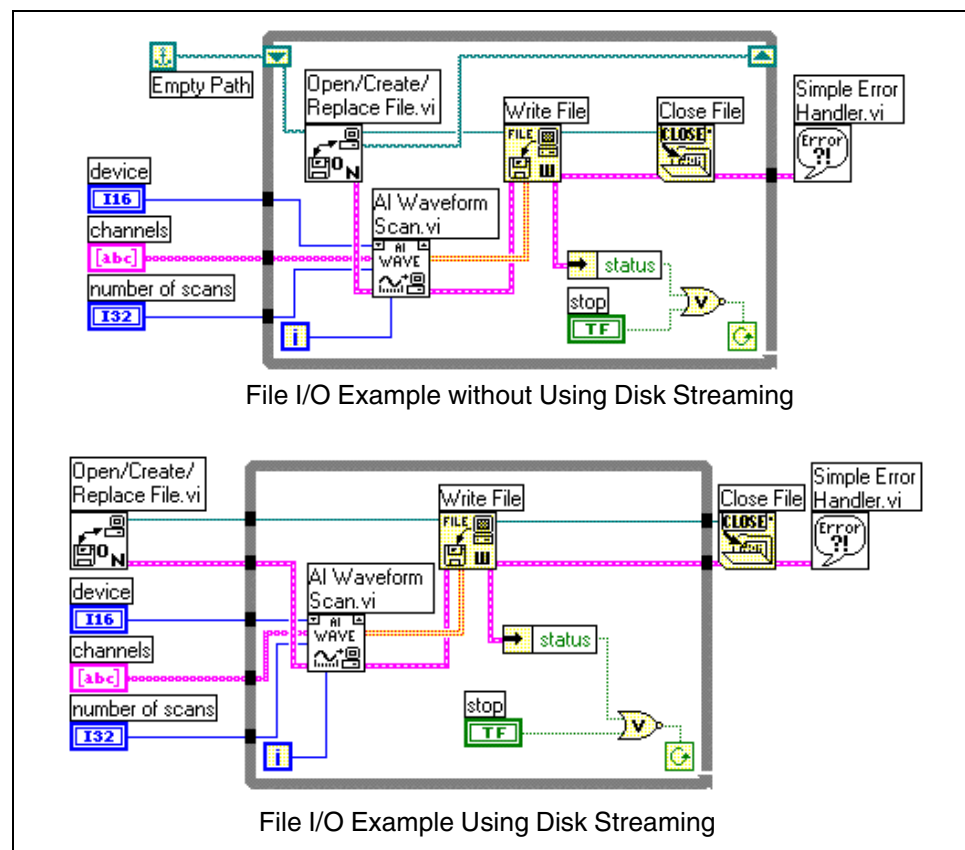
4. After you run the VI and test it, close it. Do not save any changes.

End of Exercise 4-4

C. Streaming Data to Disk

When your application repeatedly acquires data and writes it to disk, you can improve the efficiency of the file I/O operations if you do not open and close the file each time you access it. The technique of leaving files open between the write operations is called disk streaming. Disk streaming is a technique we have used in this course without describing its performance advantages. Refer to Exercise 4-3 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.



Disk streaming is especially important when you use the high-level VIs, such as Write To Spreadsheet File and Write Characters To File. These VIs open, write, and close the file each time they run. Thus, if you call one of these VIs in a loop, you perform unnecessary Open File and Close File operations during every iteration of the loop.

Summary, Tips, and Tricks

You can use the LabVIEW file I/O 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 are files in which all data is stored 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 will 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 are files in which data is stored 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. We recommend you 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. We recommend using 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 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

- 4-5 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. Name the VI `File Writer.vi`.
- 4-6 Write a VI that uses the Advanced File I/O functions to read the file created in Exercise 4-5. Name the VI `File Reader.vi`.



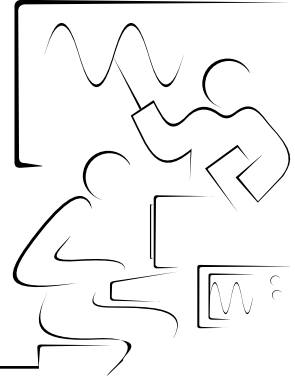
Tip Use the EOF function located on the **File I/O»Advanced File Functions** palette to obtain the length of the written file.

Notes

Notes

Lesson 5

Developing Larger Projects in LabVIEW



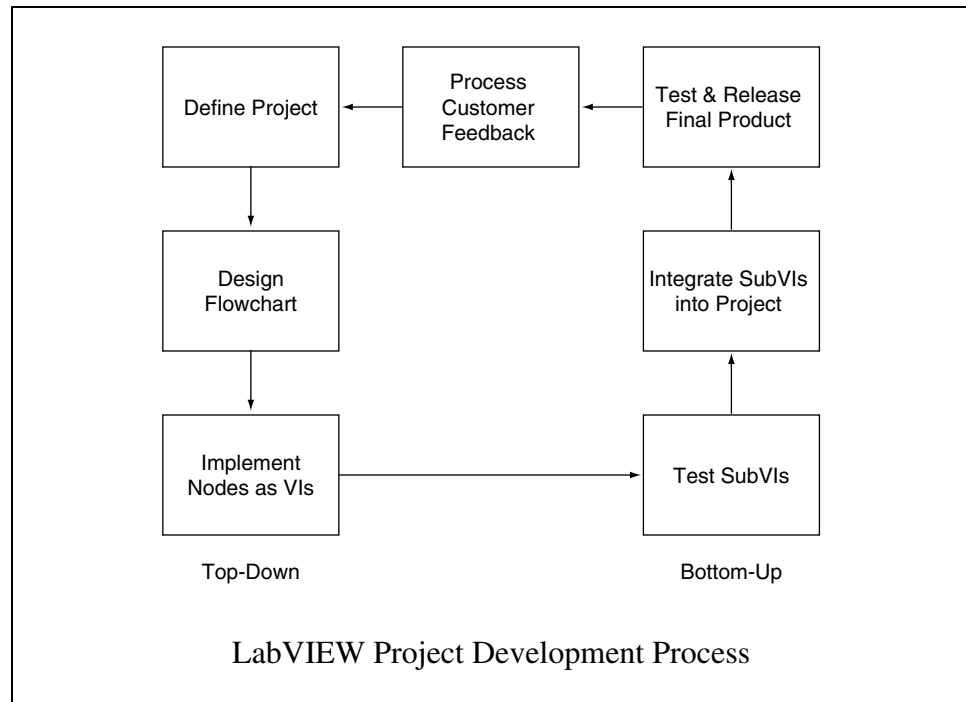
This lesson describes some of the issues involved when building larger LabVIEW projects, including the design process, the organization of subVI components, and the process of combining those components to create a complete application.

You Will Learn:

- A. How to assemble your LabVIEW application from developed subVIs.
- B. The LabVIEW features for managing project development.
- C. About LabVIEW tools for project management.

A. Assembling a LabVIEW Application

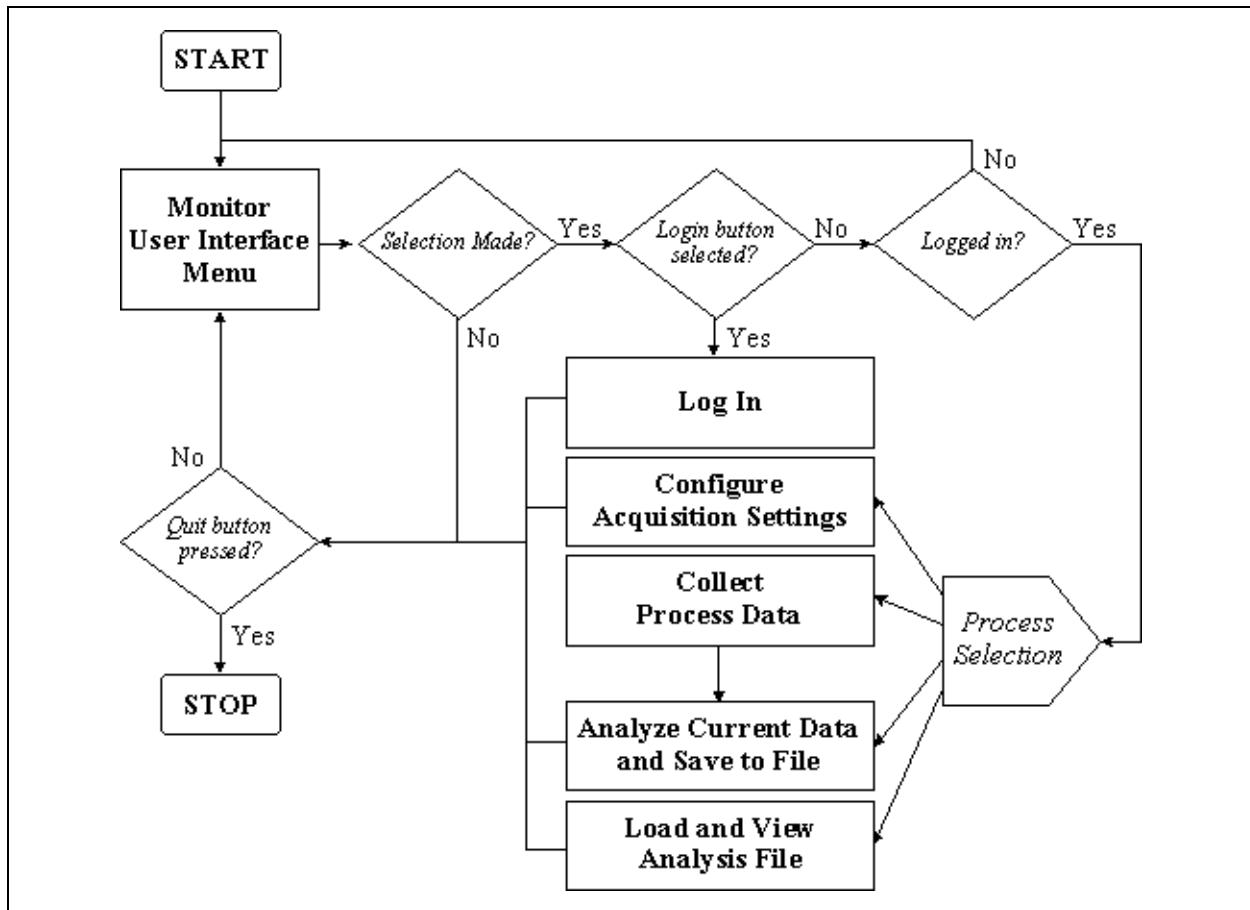
Lesson 1 of this course described a general approach to developing LabVIEW applications. This approach involved a top-down design of an application, followed by a bottom-up implementation of the project as a series of subVIs. Following is a review of this development cycle.



In this lesson, you will put the VIs built in previous lessons into a large application. You will develop a generic 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.
- If the user is not correctly logged in, other features are disabled.
- 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.
- As soon as the data have been acquired, and any time the user selects thereafter, the user can 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 previously saved to disk.
- Stops the application with the click of a **Stop** button.

In Lessons 1 through 4, you concentrated on the bottom elements of the development process, implementing and testing subVIs which correspond to nodes of the following project flowchart.



Developing larger applications in LabVIEW is a different process than building the relatively straightforward, mostly single-task VIs we have been working with so far. As you develop larger applications, you will find that working in teams, controlling source code, and creating an intuitive hierarchy scheme for your application's subVIs increases productivity and improves documentation and application performance.

Teamwork

Business management experts stress the importance of teamwork in designing solutions for large, complex tasks. When using LabVIEW, teamwork is a powerful tool for increasing your productivity in graphical programming. When several people work together to develop a project in LabVIEW, it is essential that all team members agree on a method of source code control and organization.

Source Code Control

As the number of VIs used in your project grows, your development team should take measures to control the source code of the project. You can use the built-in VI Revision History window, located in **Tools»VI Revision History**, to record changes you make to VIs. Then, you can set up a check-out system for the VIs.

Third-party development tools that offer a check-out system to protect project files typically will work with LabVIEW, as long as they can manipulate binary files. If you use a third-party tool to manage your project's files, remember that VIs and LLBs, VI libraries, are binary files. Therefore, tools that merge source files do not work correctly with LabVIEW files.

If you use a check-out system and VIs are stored in an LLB, you must check out the entire library of VIs. Keep this in mind as you work on a project. One VI management technique is to edit a copy of the VI you want to modify and lock the original VI to indicate to others that the VI should not be modified. To lock a VI, enable the Locked setting by selecting **Security** from the top pull-down menu in the **File»VI Properties** dialog box. You should also indicate that you have checked out the VI by adding a comment in the VI Revision History window. When you lock a VI, its block diagram cannot be modified. After you finish modifying the VI, replace the VI stored in the LLB with the new version and unlock it so others know it is safe to check out the VI.

To document your VIs, select **Description and Tip** from the top pull-down menu in the **File»VI Properties** dialog box. Control descriptions also provide online help you can use in the final product.

Using LabVIEW Libraries (LLBs)

You can load and save VIs to/from a special file called a VI library (.llb). VI library files are useful for organizing VIs. Some of the advantages of using LLBs are:

- You can use names up to 255 characters in length for your VIs.
- LLBs store several VIs in a compressed format, which conserves disk space.
- You can tag VIs to be top level within a library.
- Porting VIs to another platform is simplified because you do not need to transfer as many files.

There are also some disadvantages when using LLBs:

- Loading and saving time for applications that use a large number of subVIs stored in LLBs is generally slower than if the subVIs are stored directly on disk using directories.
- VI libraries are not hierarchical. You cannot create a VI library within another VI library, so all VIs and subVIs are at the same level.
- When LLB files become large, that is, up to several megabytes, saving edits to a VI stored in a LLB takes longer.
- Recall that the operating system views an LLB as a single file, so when you save a VI in an LLB, LabVIEW and the operating system must manipulate a very large file. When working with such large files, you run a higher risk of running out of memory or disk space when manipulating the file, increasing the likelihood of corrupting the library and losing work.
- You cannot use your operating system's file searching tools to locate VIs stored in a LLB.
- Source code control of VIs stored in LLBs is more difficult.

As a rule of thumb, try to limit the size of your LLB to roughly 1 MB. Whether you store your VIs in an LLB or directly on disk can depend on file naming restrictions in your operating system or how you select to organize your files.

Refer to the *LabVIEW Development Guidelines* manual, available at ni.com, for more information about designing, building, and testing applications in LabVIEW complete with a style guide, checklists, and references.

Creating LabVIEW Applications

You are now going to create the application described on the previous few pages. The first step in creating a professional, stand-alone application with VIs is to understand the architecture of your application. You are going to use the *State Machine VI Architecture* described in Lesson 1 for this application. This application contains a single top-level VI that calls all other VIs as subVIs.

In the next four exercises you will build the Application Exercise VI. Each exercise adds a state or two to the application. This method is used so you can incrementally test the application as it is being built.

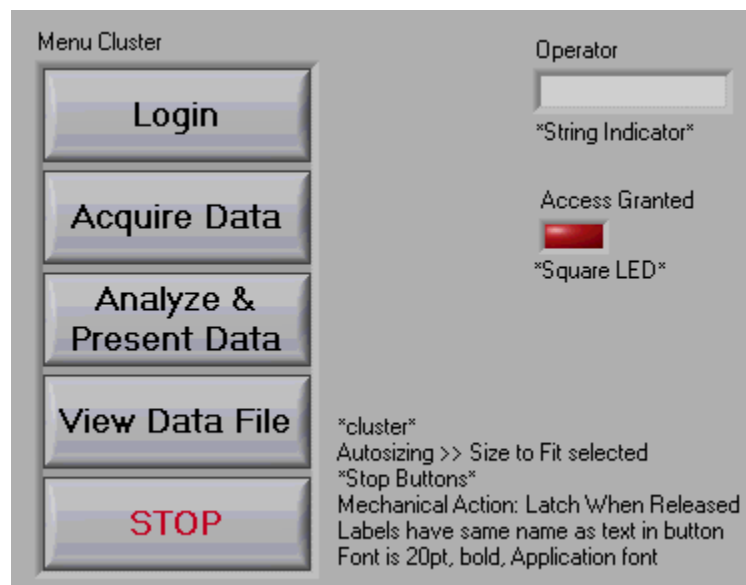
Exercise 5-1 Application Exercise (5-1) VI

Objective: To use the Login VI created in Exercise 3-1 to provide password security to an application.

You will begin creating an application that uses several of the VIs you wrote earlier in the course. The first stage of this development is to add a user login procedure to the Menu VI created in Exercise 2-3.

Front Panel

1. Open the Menu VI in the `exercises\Basics2` directory. You created this VI in Exercise 2-3.

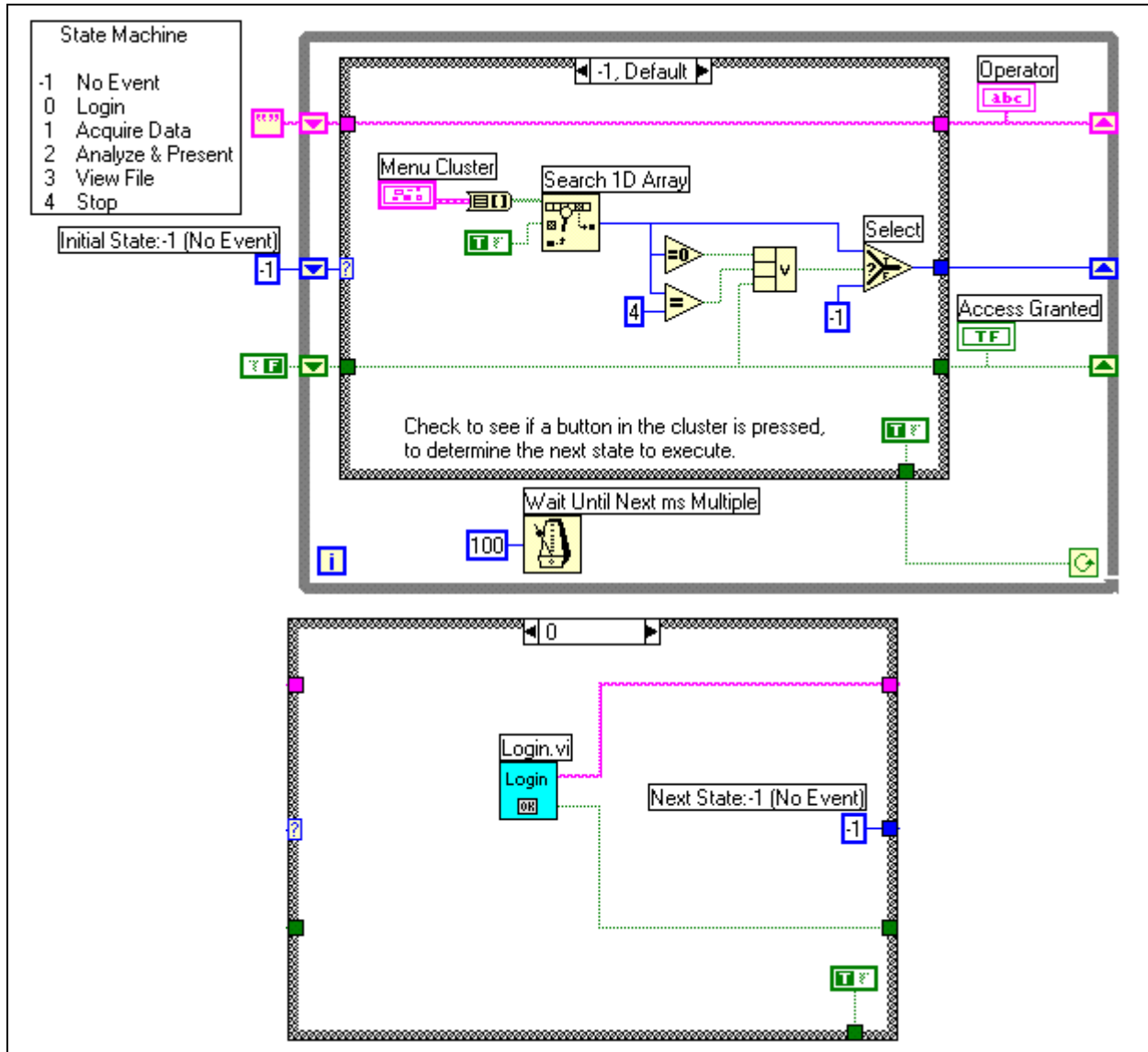


Note If you did not complete Exercise 2-3, locate Menu VI in the course solutions stored on your computer.

2. Add a String indicator located on the **Controls»String & Path** palette on the front panel. Label it **Operator**.
3. Add a Square LED located on the **Controls»Boolean** palette on the front panel and label it **Access Granted**. You will write this VI so that until access is granted by the user logging in with a correct name and password, the user can only use the **Login** and **Stop** buttons.
4. Open the block diagram.

Block Diagram

1. Modify the following VI. Only modify the While Loop and Cases -1 and 0. The Login VI is called when the user presses the **Login** button on the front panel (Case 0 of the Case structure). Remember to delete the One Button Dialog function that you placed in Case 0 in Exercise 2-3.



Note If you did not complete Exercise 3-1, locate Login VI in the course solutions stored on your computer.

2. In addition, modify the VI so that:
 - a. The Operator indicator is initialized to an empty string when the VI starts.
 - b. If the user has not logged in with a correct name and password, only the **Login** and **Stop** menu options execute.
 - c. If the **Login** VI returns a value of FALSE for the access granted output, the Operator indicator should show an empty string.



Note The VI needs the Operator and Access Granted information in subsequent loop iterations. Therefore, this VI uses two more shift registers to store the data. You will need to make sure that the other cases in the Case structure pass the data through correctly, straight through the cases.

By using the shift registers, notice that only Case 0, in which the Login VI runs, can change the Operator and Access Granted values.

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.

3. Run the VI and test it to make sure it behaves properly. Use the LabVIEW debugging tools—execution highlighting, single-stepping, probes, and breakpoints—to determine the dataflow of the VI.
4. Save the VI as `Application Exercise (5-1) .vi`. Do not close the VI.

End of Exercise 5-1

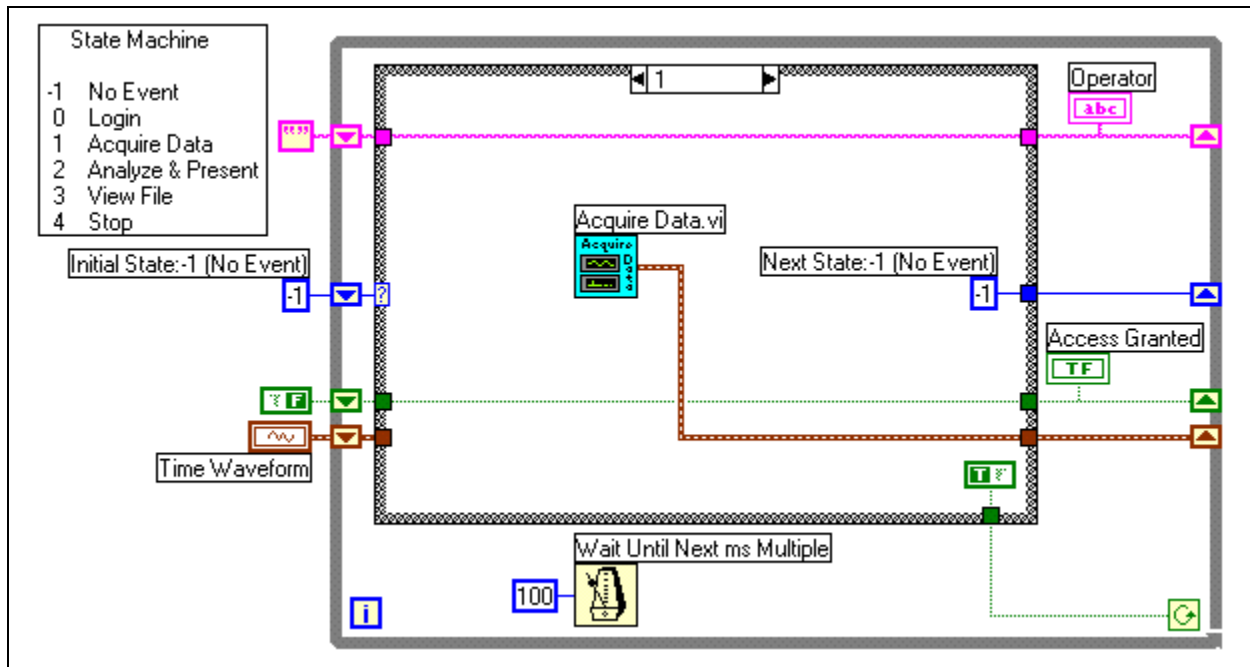
Exercise 5-2 Application Exercise (5-2) VI

Objective: To add the Acquire Data VI to the application started in Exercise 5-1.

In the previous exercise, you began building the Application Exercise VI. Now, you will add the Acquire Data VI, which you built in Exercise 2-2.

Block Diagram

1. Make sure that the Application Exercise (5-1) VI is open. You will not modify the front panel in this exercise. Open the block diagram.



2. Show Case 1 of the Case structure and delete the One Button Dialog function. Add Acquire Data VI, which you built in Exercise 2-2. If you did not finish that exercise, use the VI from the solutions located on your computer.
3. Add a shift register to the border of the While Loop. Connect the **Time Waveform** output of the Acquire Data VI to the right side of the shift register.
4. You must initialize the new shift register you created.
 - a. Right-click the left side of the shift register that contains the configuration cluster and select **Create»Control** from the shortcut menu. This creates an empty waveform control on the front panel.
 - b. Hide this Time Waveform control by right-clicking the terminal and selecting **Hide Control** from the shortcut menu. This makes the waveform control invisible so it does not confuse users.

5. Modify the VI Properties of the Acquire Data subVI so that it appears like a dialog box when it is called. Double-click the Acquire Data subVI to open its front panel. Select **File»VI Properties** and select **Window Appearance** from the top pull-down menu and click the **Dialog** button. Click **OK** then save and close the Acquire Data VI.
6. 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 the VI so that the data pass through the other cases unchanged, recall the method used in Exercise 5-1.
7. Save the VI as `Application Exercise (5-2) .vi`. Run it and test it.

End of Exercise 5-2

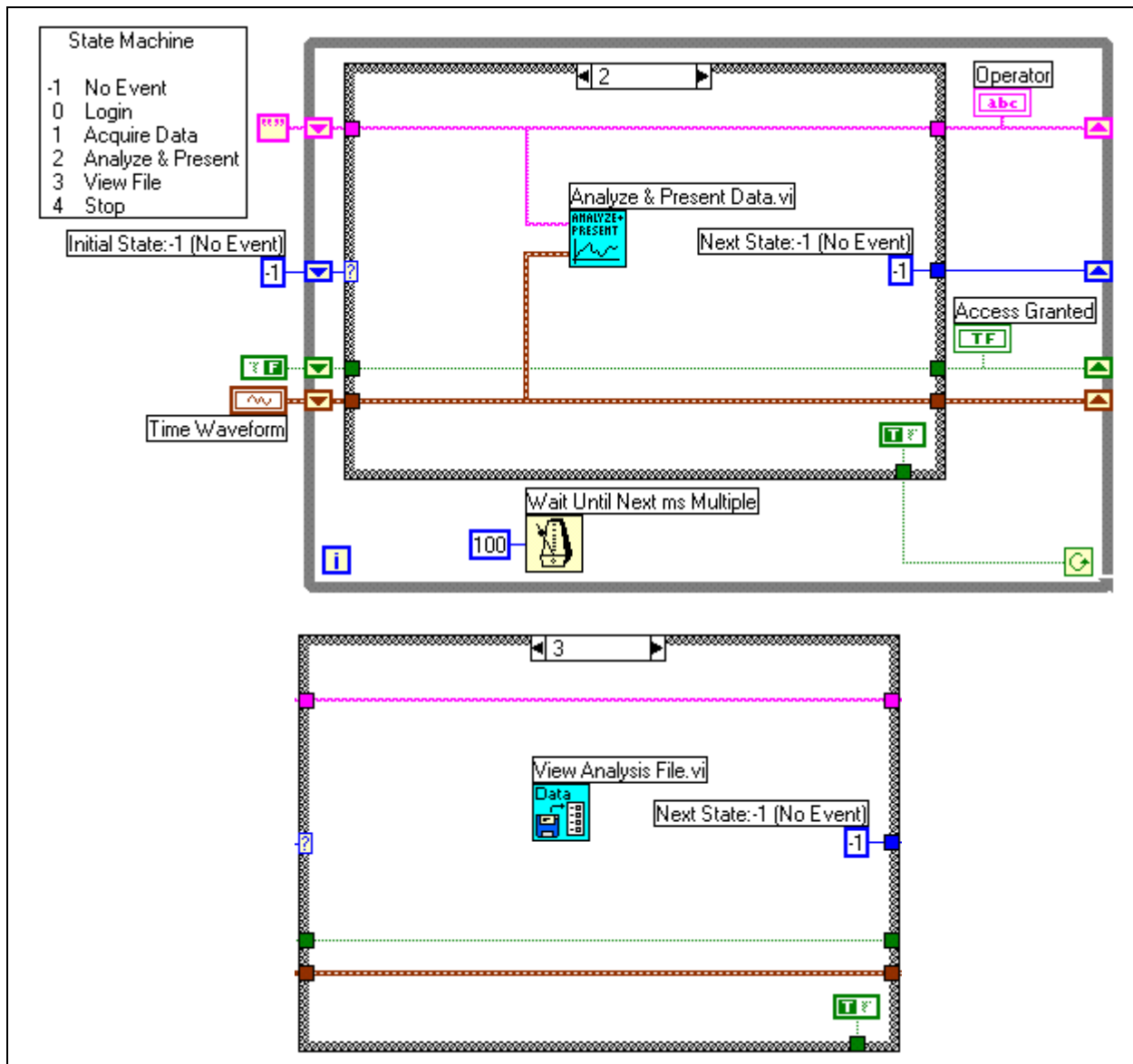
Exercise 5-3 Application Exercise (5-3) VI

Objective: To finish the Application Exercise VI by adding the subVIs for the Analyze & Present Data and View Analysis File buttons.

You will add the Analyze & Present Data VI from Exercise 2-7 and the View Analysis File VI, which you studied in Exercise 4-4.

Block Diagram

1. Make sure that the Application Exercise(5-2) VI is open. You will not modify the front panel in this exercise. Open the block diagram.



2. Delete the One Button Dialog function in Case 2 of the Case structure. Add the Analyze & Present Data VI to this case. You completed this VI in Exercise 2-7.
 - a. Connect the string containing the operator name to the **Operator** input.
 - b. Connect the waveform containing the collected data to the **Data** input.
3. Show Case 3 of the Case structure and delete the One Button Dialog function. Add the View Analysis File VI. You do not need to make any connections to it.
4. Save the VI as `Application Exercise (5-3) .vi` and run it. You should now be able to perform all of the options available in the menu after you log in with a valid name and password.



Note You must select a subset of the acquired data when you select **Analyze & Present Data**. After you select the data subset, click the **Analyze** button and then select the button to save the data to file. Otherwise, the file does not contain any waveform data.

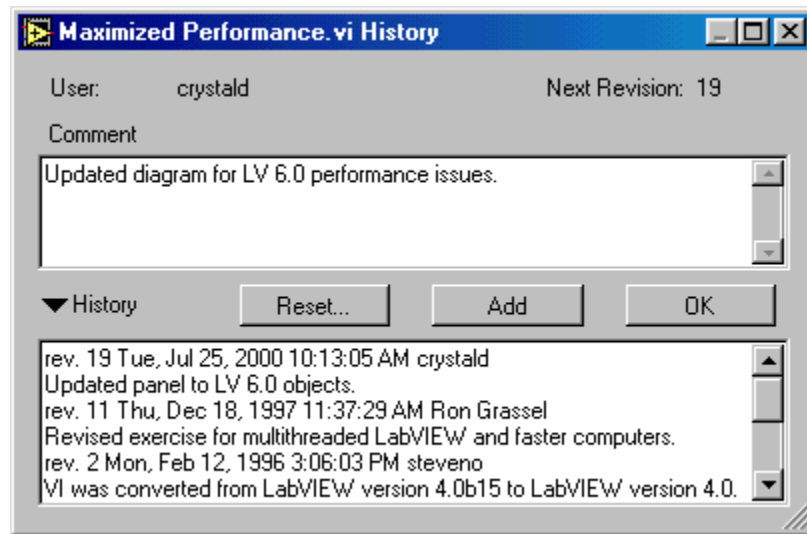
End of Exercise 5-3

B. LabVIEW Features for Project Development

VI History

One of the most useful LabVIEW tools for team-oriented development is the VI Revision History window. Every VI has a History window that displays the comments recorded by those who worked on the VI. You can use the VI Revision History window to track changes and revisions to VIs and applications. You can configure LabVIEW with several different VI Revision History options, which apply on a global or per-VI basis, using the LabVIEW preferences in **Tools»Options** or **File»VI Properties**.

To open the Revision History window for a VI, select **Tools»VI Revision History**.



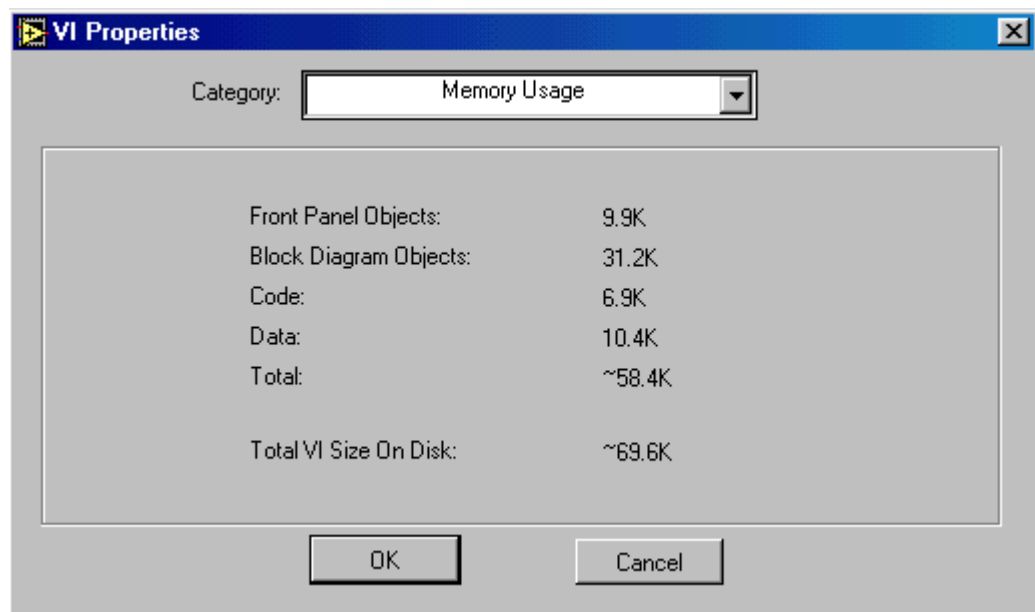
After typing your comments into the Comment area of the Revision History window, click **Add** to add them to the VI's history. The VI Revision History window also keeps track of revision numbers. Each time you save the VI, the revision number for the VI increments by one.

In addition to the revision number and the date and time at which a comment is added to a VI's history, the current user's name appears in the **User** field. You can configure the LabVIEW preferences to prompt for a user name at launch time, by selecting **Revision History** from the top pull-down menu in the **Tools»Options** dialog box, or you can change the user name by selecting **Tools»User Name**.

VI Hierarchy

One of the most important advantages of breaking 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. Hierarchical applications are easier to develop, read, document, and modify.

Therefore, as a general rule, it is recommended that you 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 block diagram, select **Memory Usage** from the top pull-down menu in the **File»VI Properties** dialog box. Typically, you should consider breaking a VI into several subVIs if the block diagram for your VI is too large to fit entirely on the screen.

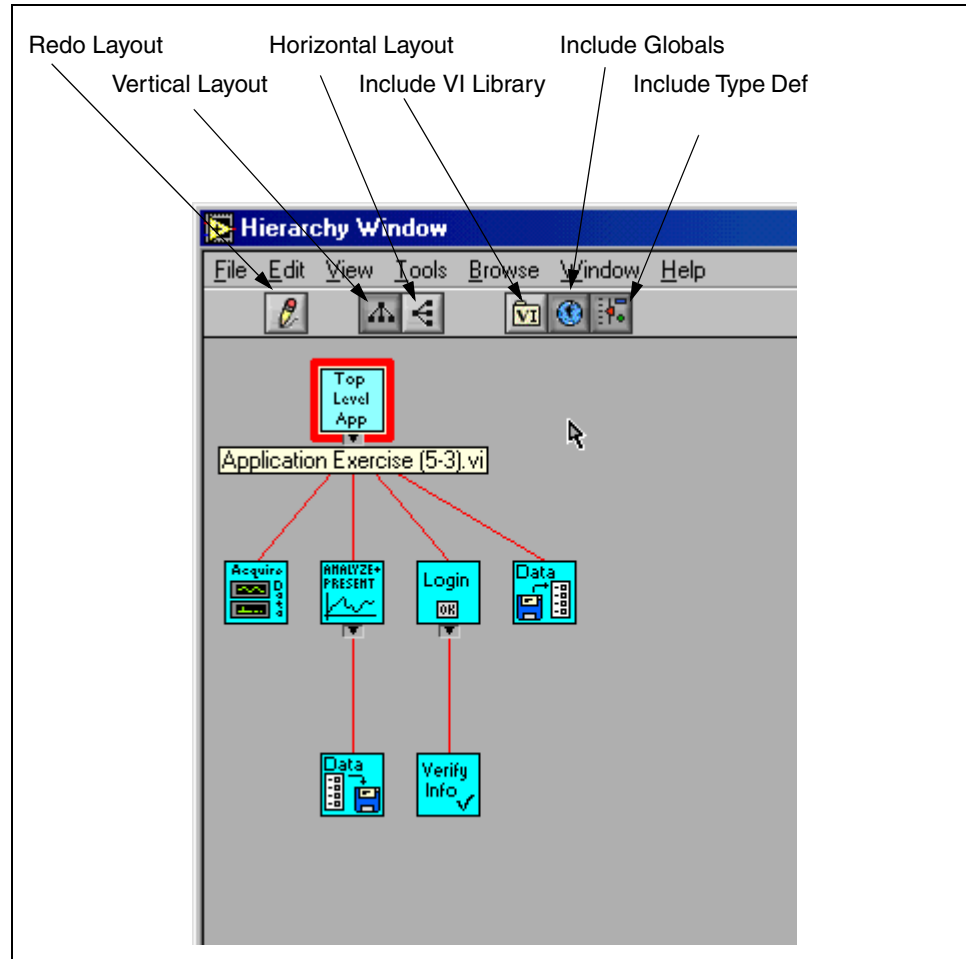


If you find that the block diagram for a VI is getting too large, you can convert part of it into a subVI by using **Edit»Create SubVI**. This capability gives you a fast and easy method to create your application's VI hierarchy as you develop the source code.

The VI Hierarchy window is also a valuable tool for locating subVIs and viewing the overall layout of the project as your application grows. To view the Hierarchy window, select **Browse»Show VI Hierarchy**. A new window appears showing the hierarchies of all top-level VIs in memory.

You can use the options available in 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. Clicking the small arrow appearing next to a VI expands or collapses the view of that VI's hierarchy. Thus, you can expand different branches of the overall hierarchy.

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. The entire hierarchy is shown by right-clicking a blank area of the window and choosing **Show All VIs**.



As you move the cursor over the icons shown in the Hierarchy window, the name of the VI follows the VI. You can double-click an icon to open the VI. You can also locate a VI in the hierarchy by typing in the name of a VI while in the Hierarchy window. As you type the name, the Hierarchy window scrolls to the appropriate VI. You can also use the **Find** feature to search the Hierarchy window for a VI.

The Hierarchy window can also be used 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 will build the basic application hierarchy, which will now appear in the Hierarchy window. You can then begin to develop each subVI,

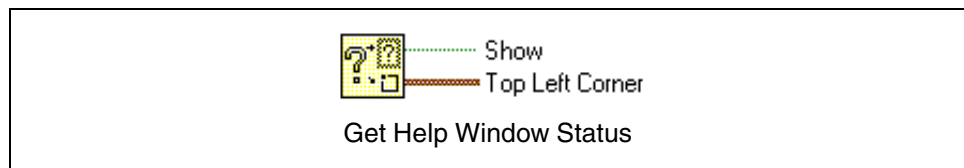
perhaps color-coding their icons, which will also be colored in the Hierarchy window, to reflect their status. For example, white icons could represent untouched VIs, red icons could represent subVIs in development, and blue icons could represent completed VIs. While this is only one example of using the Hierarchy window as a development tool, it demonstrates the usefulness of this window for organizing a project.

Using Online Help in Your LabVIEW Applications

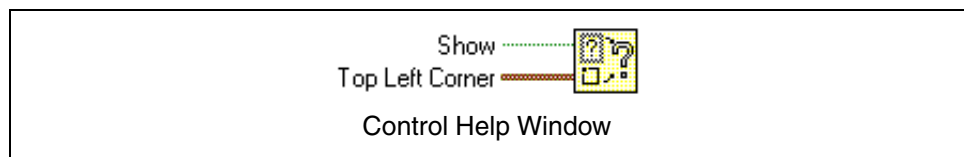
As you put the finishing touches on your application, you might want to provide online help to the user. LabVIEW offers several mechanisms for doing this. By using the **Description and Tip** option available on every front panel control and indicator, you create not only well-documented VIs, but also VIs that have extensive online help available to the user.

If you select **Help»Show Context Help**, the Context Help window appears. As you move the cursor over an object, the Context Help window updates to show its description. You can programmatically show or hide the Context Help window with the **Get Help Window Status** and **Control Help Window** functions, which you find in the **Functions»Application Control»Help** palette.

Use this function to determine if the Context Help window is visible and the location of the upper left corner of the window.



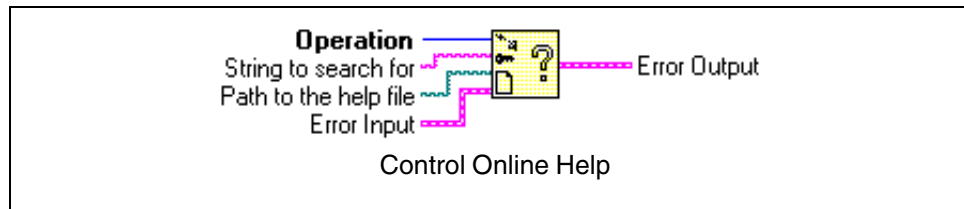
With this function, you can control whether the Context Help window is visible, and where it will appear when shown.



You can also use the Control Online Help function to access the *LabVIEW Help* or custom help files that you compile using third-party tools. The type of online reference development tools you can use to develop this type of help file depend on the platform on which your application will run.

- **Windows** **Microsoft Help (.hlp files)**
- **Macintosh** **QuickView**
- **Sun/HP-UX** **HyperHelp**

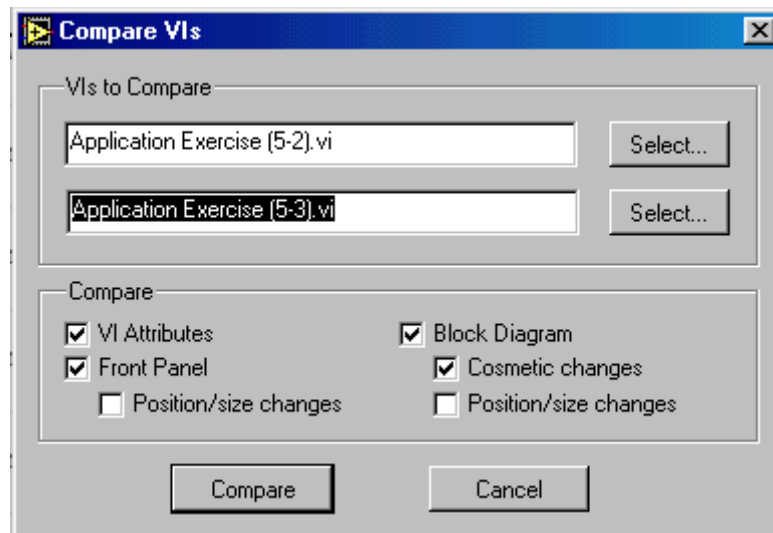
With this function, you manipulate an online help file. You can list the contents of the help file, jump to keywords in the file, or close the online help file.



Providing online help and reference materials for your application makes it easier to use and gives it a more polished, professional look.

VI Comparison

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.



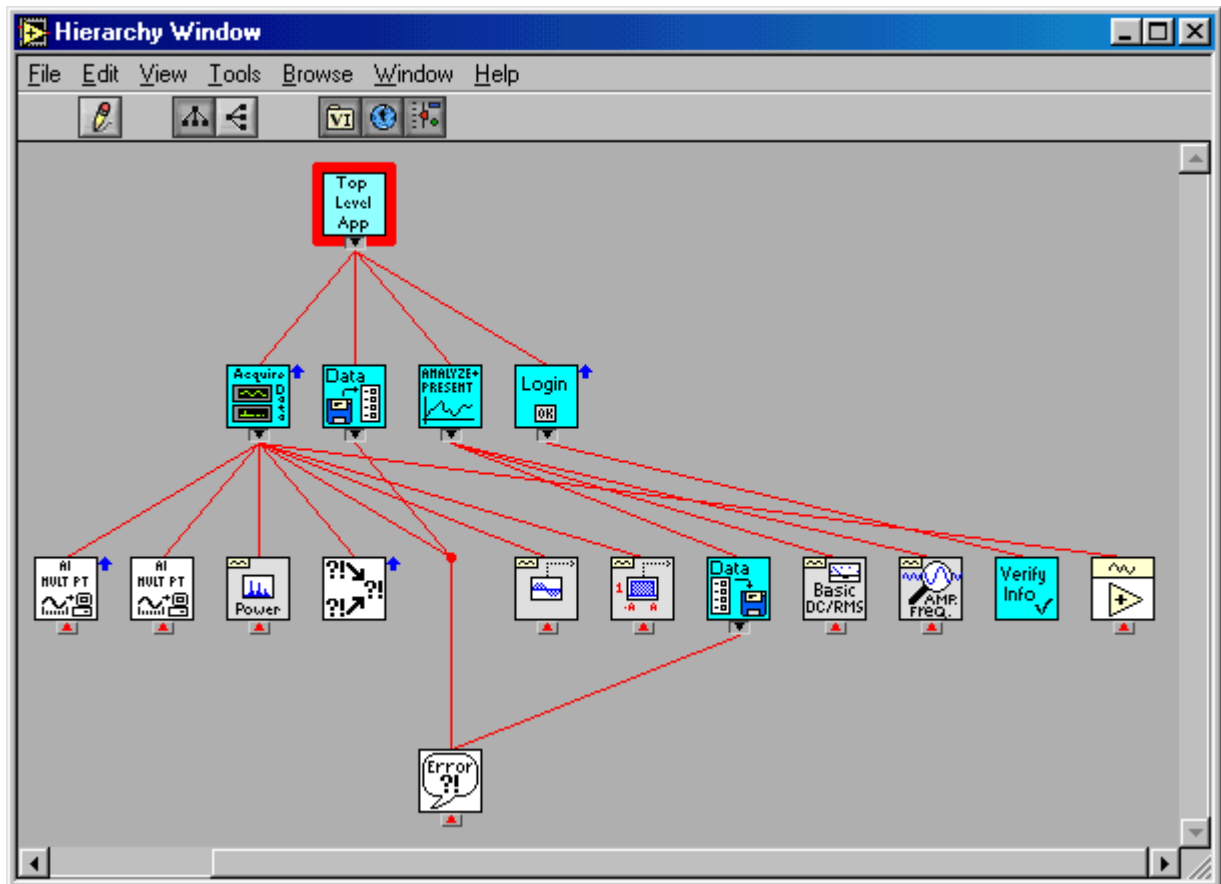
From this dialog box, you can select the VIs you want to compare, as well as the characteristics of the VIs to check. When you compare the VIs, both VIs will be displayed, along with a Differences window that lists all differences between the two VIs. In this window, you can select various differences and details to view, which can be circled for clarity.

Exercise 5-4

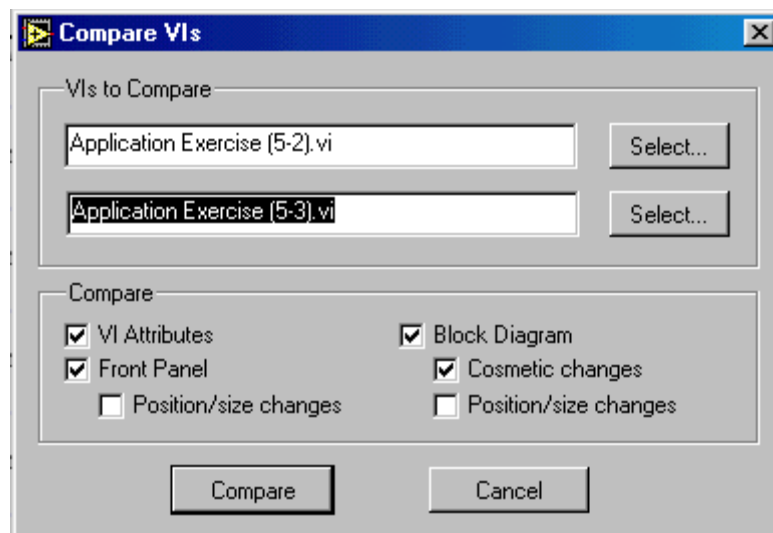
Objective: To examine some of the built-in LabVIEW features for handling applications.

In this exercise, you will explore some of the features built into LabVIEW for handling applications.

1. Open the Application Exercise(5-3) VI you created in the previous exercise. Close any other VIs loaded into memory.
2. Select **Tools»VI Revision History**. The history window for the VI should appear.
3. Click **Reset** in the Revision History window to clear the current history. Click **Yes** to confirm the deletion of the history and resetting of the revision number.
4. In the Comment box of the History window, type in `Initial Application Created.` and then click the **Add** button. Your comment should appear in the Revision History listing, along with a date and time stamp. Close the Revision History window.
5. Select **Browse»Show VI Hierarchy**. The application's 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 Application Exercise(5-2) VI you completed in Exercise 5-2, change to the front panel of the Application Exercise(5-3) VI, and then select **Tools»Compare»Compare VIs** to display the **Compare VIs** dialog box.



10. Using the **Select** option, make sure that the two Application Exercises are listed in the **VIs to Compare** box, and that the **Compare** options are set as in the previous example.
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 Application Exercise (5-2) .vi.

End of Exercise 5-4

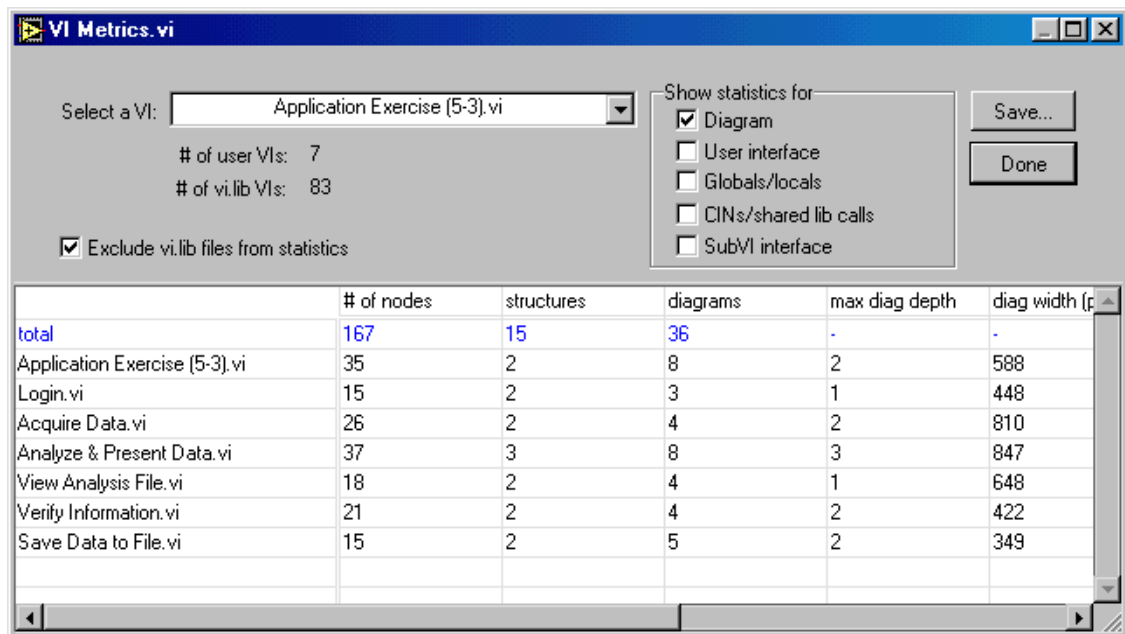
C. LabVIEW Tools for Project Management

There are several tools you can add to the LabVIEW environment to assist in project development. The following section describes two of these tools that are included in the LabVIEW Professional Development System.

Source Code Controls Tools

These tools simplify management of larger applications. The first feature added with these tools is a basic source code control (SCC) system that is tightly integrated into the LabVIEW environment. This system contains many features found in third-party source code control systems, such as file check-in/out, revision tracking, and support for multiple users. In addition, the SCC tools can support third-party source code control systems such as Microsoft Visual SourceSafe. As mentioned earlier in this lesson, however, VIs for a project must be saved as individual files (instead of in LLBs), because the file management features of the various operating systems do not support LabVIEW libraries.

The SCC tools also contain a VI Metrics tool to measure the complexity of an application similar to the widely used Source Lines of Code (SLOCs) metrics for textual languages. With the VI Metrics tool, you can view statistics about VIs, such as the number of nodes (functions, subVI calls, structures, terminals, and so on) of a VI and its subVIs. Other statistics include the number and sizes of block diagrams, number and type of user interface objects, number of accesses to global and local variables, and number of calls to Code Interface Nodes or shared libraries. You access the following VI Metrics window from the **Tools»Advanced** menu.

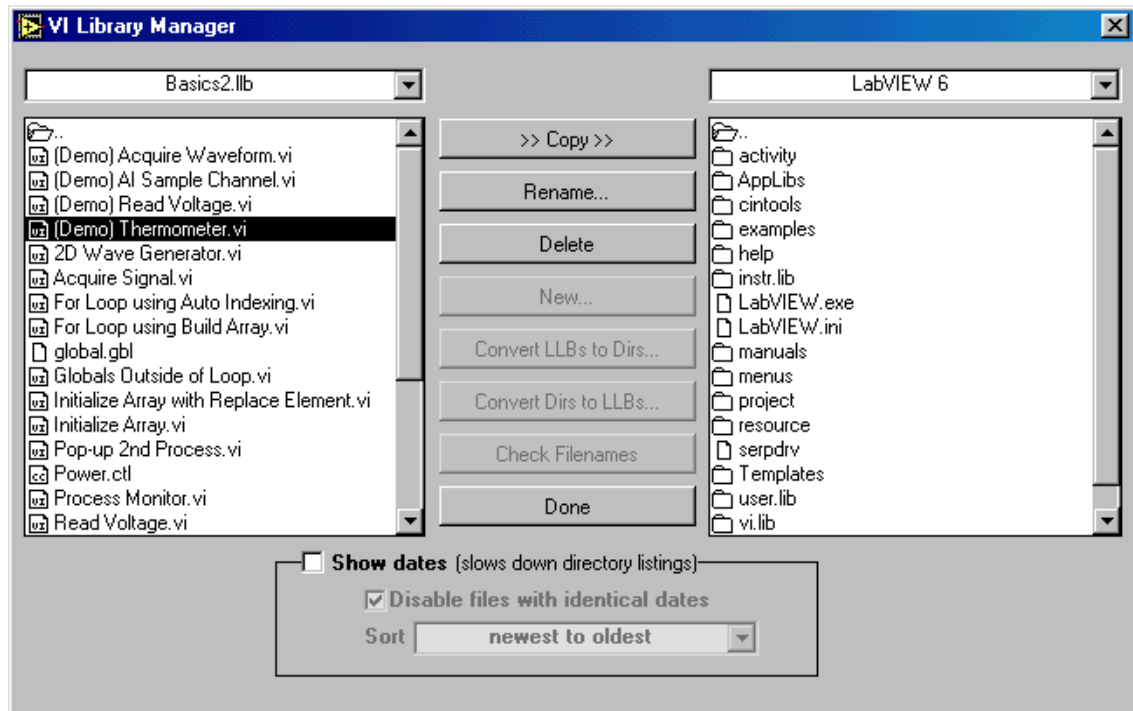


The screenshot shows the VI Metrics window with the following settings and data:

- Select a VI: Application Exercise (5-3).vi
- # of user VIs: 7
- # of vi.lib VIs: 83
- Exclude vi.lib files from statistics
- Show statistics for:
 - Diagram
 - User interface
 - Globals/locals
 - CINs/shared lib calls
 - SubVI interface

	# of nodes	structures	diagrams	max diag depth	diag width (p
total	167	15	36	-	-
Application Exercise (5-3).vi	35	2	8	2	588
Login.vi	15	2	3	1	448
Acquire Data.vi	26	2	4	2	810
Analyze & Present Data.vi	37	3	8	3	847
View Analysis File.vi	18	2	4	1	648
Verify Information.vi	21	2	4	2	422
Save Data to File.vi	15	2	5	2	349

A **VI Library Manager** tool is also included in this toolkit. This utility enables you to copy, rename, and delete VIs, whether they are located in LLBs or not. The VI Library Manager can also convert existing LLBs into files in a subdirectory, to make implementation of SCC tools easier.



Utilities to **Compare VIs** and **Compare VI Hierarchies** are also included. As described earlier, they are used to determine the differences between two VIs or hierarchies.

The *LabVIEW Help* and manuals for this toolkit not only provide reference material for the previous utilities, but also extensive discussions on management of software projects. Various development models, prototyping and design techniques, and project tracking methods are described in detail. Documentation of VIs and tips for developing clear code are described in detail as well.

LabVIEW Application Builder

You can use the LabVIEW Application Builder to create stand-alone executable programs for users without the LabVIEW development software. 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.

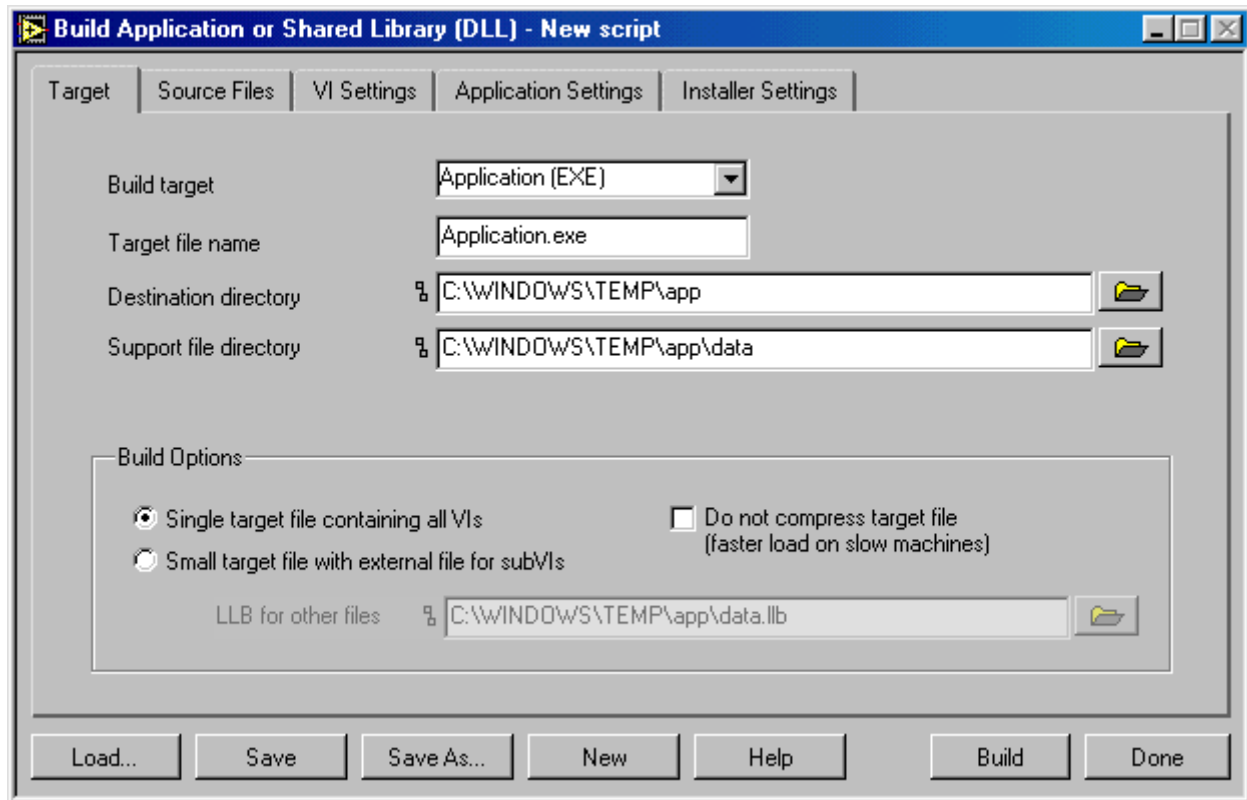
Required System Configuration

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

Turning Your Application into a Stand-Alone Executable

Before LabVIEW 5.1, the process for building an application was to save your VIs to a library, build an application using the **Build Application** dialog box, and then create an installer using the **Create Distribution Kit** dialog box. In LabVIEW 5.1, you can use the **Build Application** dialog box to do all of these operations.

When you select **Tools»Build Application or Shared Library (DLL)**, a tabbed dialog box appears. By making settings in the various tabbed pages on the dialog box, you can define the application you want to build. You can save a script and use it later to rebuild the application. The **Build Application Shared Library** dialog box contains the following tabbed pages—**Target**, **Source Files**, **VI Settings**, **Application Settings**, and **Installer** settings as shown in the following example.



- 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 on Windows.
- From the **Installer Settings** tab (**Windows** only), you create an installer. The installer is written to the directory that contains your application.

When you develop an executable VI with LabVIEW on Windows and ship it to another computer, you must also include the LabVIEW Run-Time DLL. The computer on which the VI runs must install this DLL using the LabVIEW Run-Time DLL Installer before the VI runs. If you distribute a VI using **Build Application**, the Run-Time DLL is installed automatically.



Note After the Run-Time DLL is properly installed on a machine, it can run any executable VI developed in LabVIEW. You need to include only the Run-Time DLL with the first VI sent to each computer.

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

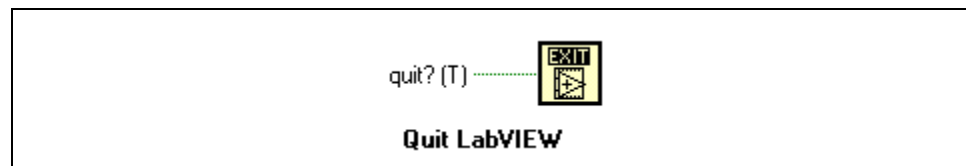
The first section of this lesson describes the architecture of your application and has you build the application in the exercises. Your application is a single top-level VI that runs when you launch the application and calls front panels from several subVIs when called. This is the most common and easiest 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 (this is described in the LabVIEW Advanced I 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 will affect the application you have currently built is that the top-level VI does not quit LabVIEW or close the VIs 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»Application Control** palette on the block diagram of the top-level VI.



Building the Application

As described earlier in this section, you use the Application Builder in LabVIEW to make either an executable or a shared library (DLL) for your application. This course describes how to build an executable and the LabVIEW Advanced I course describes 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 packaging option is to break 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 can require the presence of non-VI files to function correctly. Files commonly needed include a preferences (.ini) file for the application, the LabVIEW serpdvr file, and any help files that your VIs call. The LabVIEW serpdvr file (**Windows and UNIX**) is required for any application that uses serial port I/O. Note that run-time menu files and shared library files called using the Call Library Node 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 are also stored in the main files for the application.



Note Refer to the *LabVIEW Help* in **Help»Contents and Index** 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.

You will learn about each of the steps and options of the Application Builder when you build an application in the next exercise. After defining your build specifications using the LabVIEW Application Builder, you will save those settings to disk in the form of a build script (.bld) file.

Exercise 5-5 My Application executable

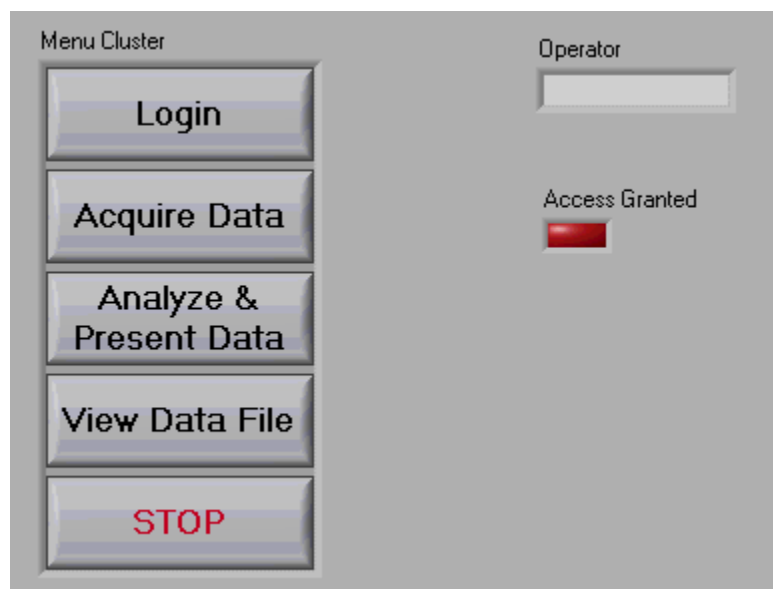
Objective: To create a stand-alone application with the Application Builder.



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 Application Exercise(5-3) VI you created in Exercise 5-3. Modify the following front panel to remove the 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. Save the VI as Application Exercise.vi.

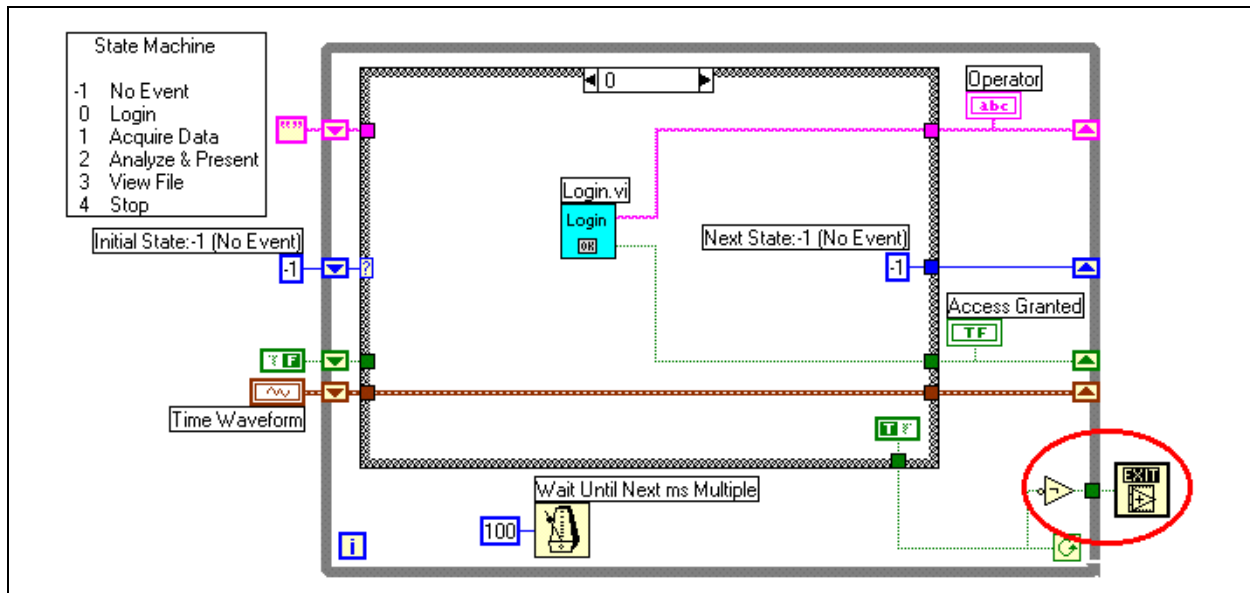
Block Diagram

4. Open the block diagram and use the following components.
 - a. Place the not function located on the **Functions»Boolean** palette on the block diagram. This function inverts the value being sent to the While Conditional terminal. When the loop ends, this value is a False, so a True value is actually sent through the loop boundary to the Quit LabVIEW function.



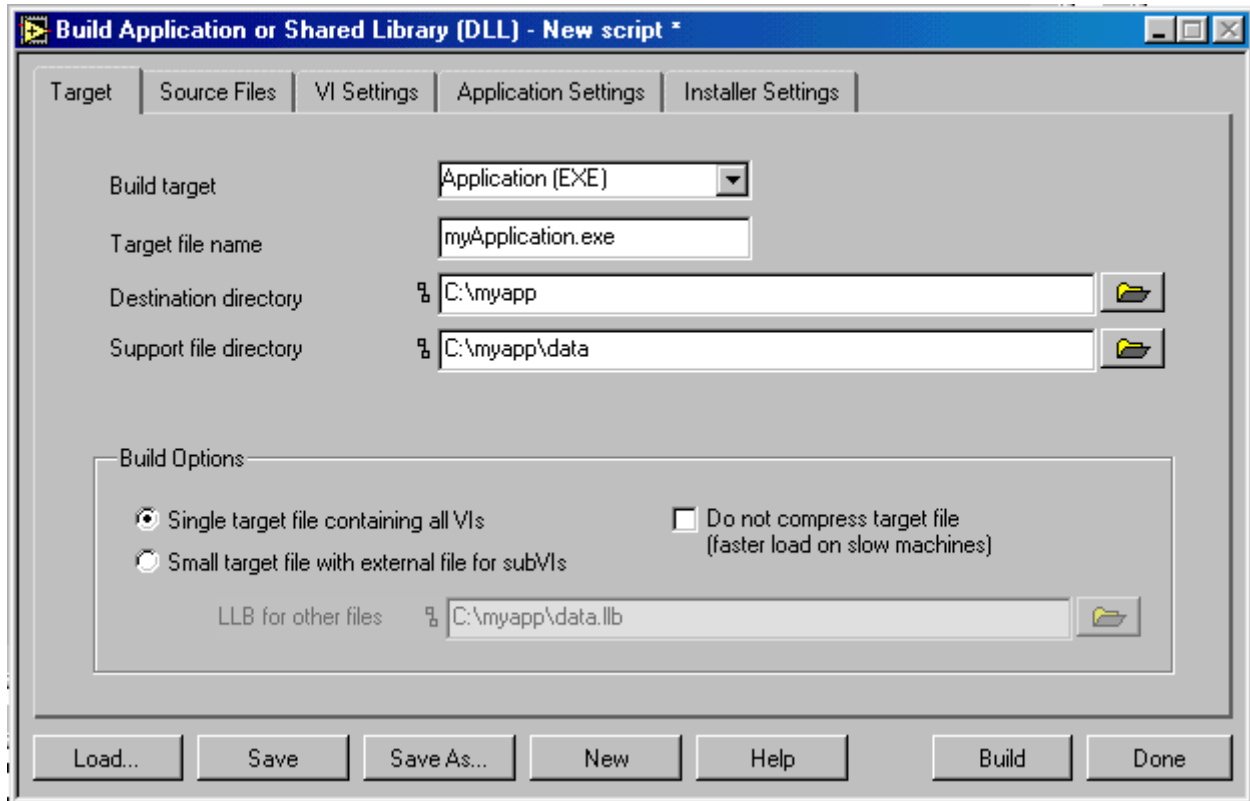


- b. Place the Quit LabVIEW function located on the **Functions» Application Control** palette on the block diagram. This function quits LabVIEW and quits the application after it has been built.

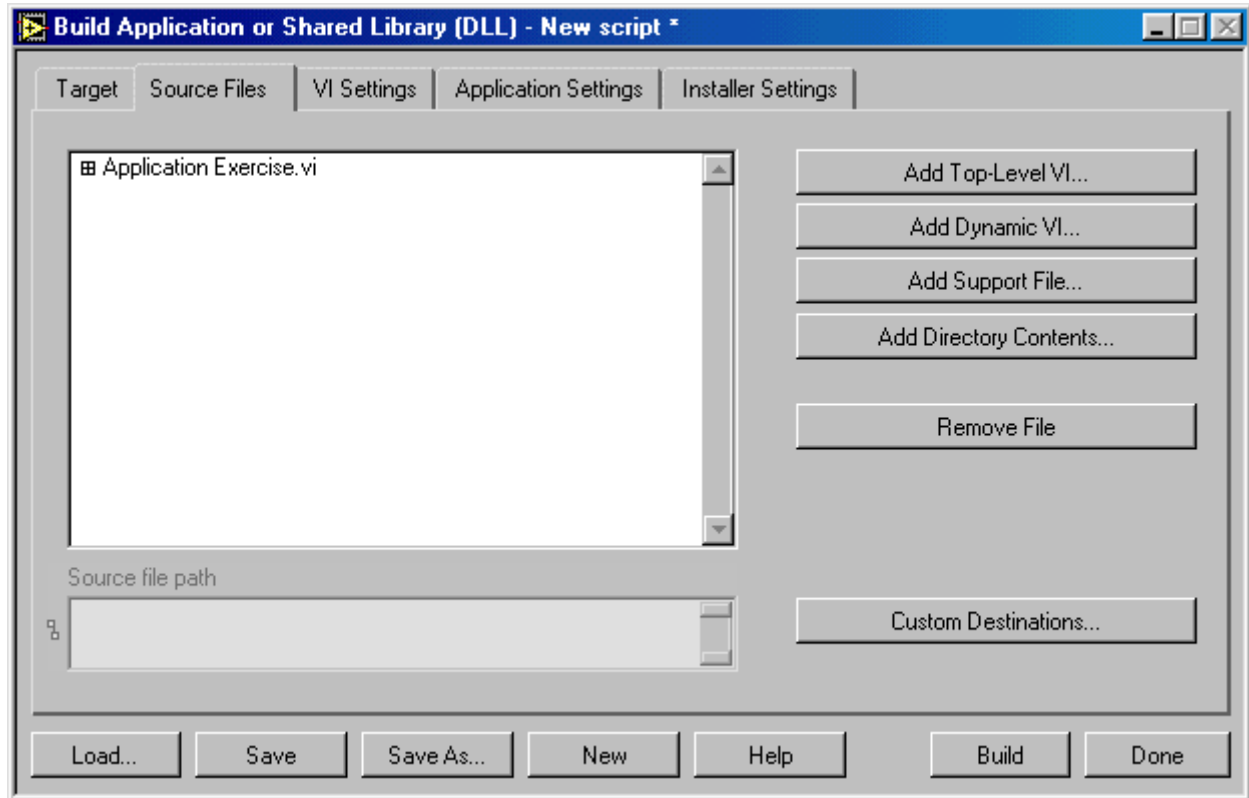


5. Save this VI as `Application Exercise.vi`.
6. Open the front panel and run the VI. When you select the **Stop** button, the VI stops and you exit LabVIEW.
7. Restart LabVIEW and open a new VI. You will not open the Application Exercise VI because the Application Builder cannot create an executable if the VIs are loaded into memory.

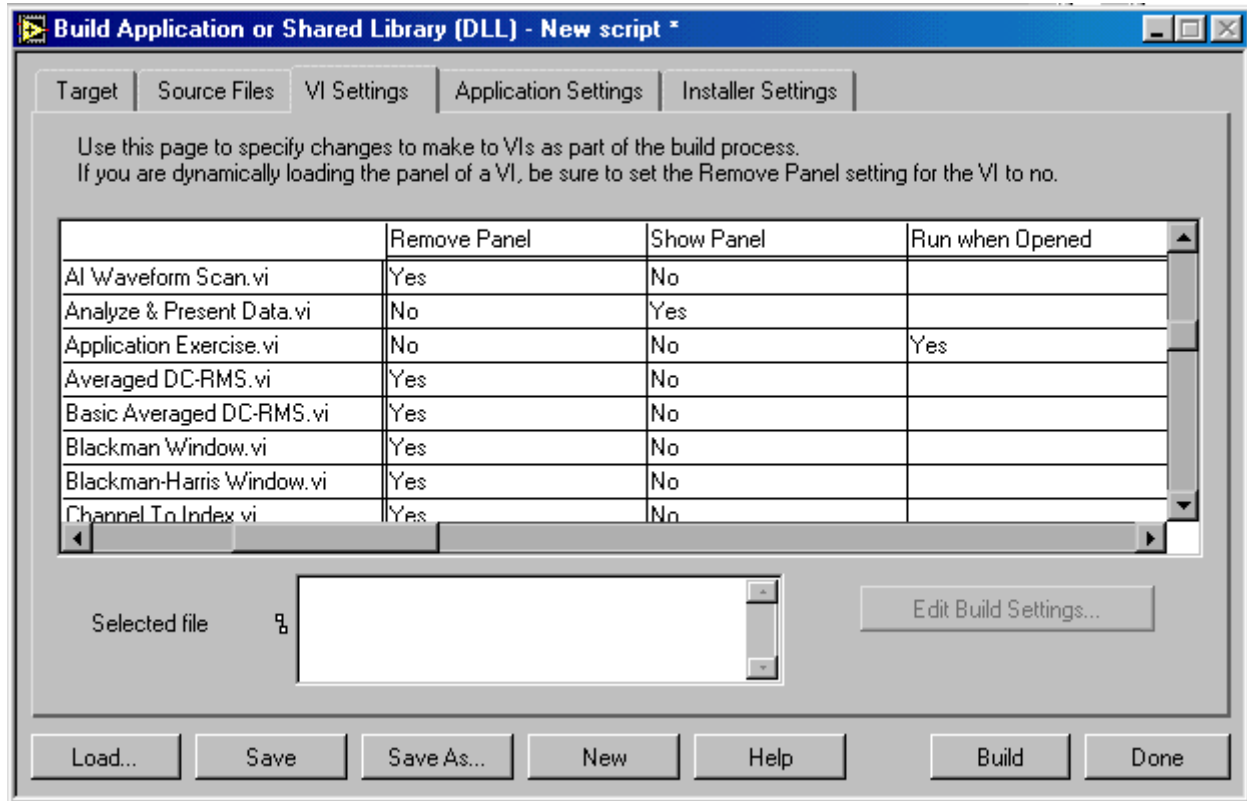
8. Select **Tools»Build Application or Shared Library (DLL)**. Build an executable from `Application Exercise.vi` called `myApplication.exe` and place it into the `C:\myapp` directory. Modify the **Target** tab as shown.



9. Click the **Source Files** tab and select the **Add Top-Level VI** button. Add Application Exercise.vi as shown.

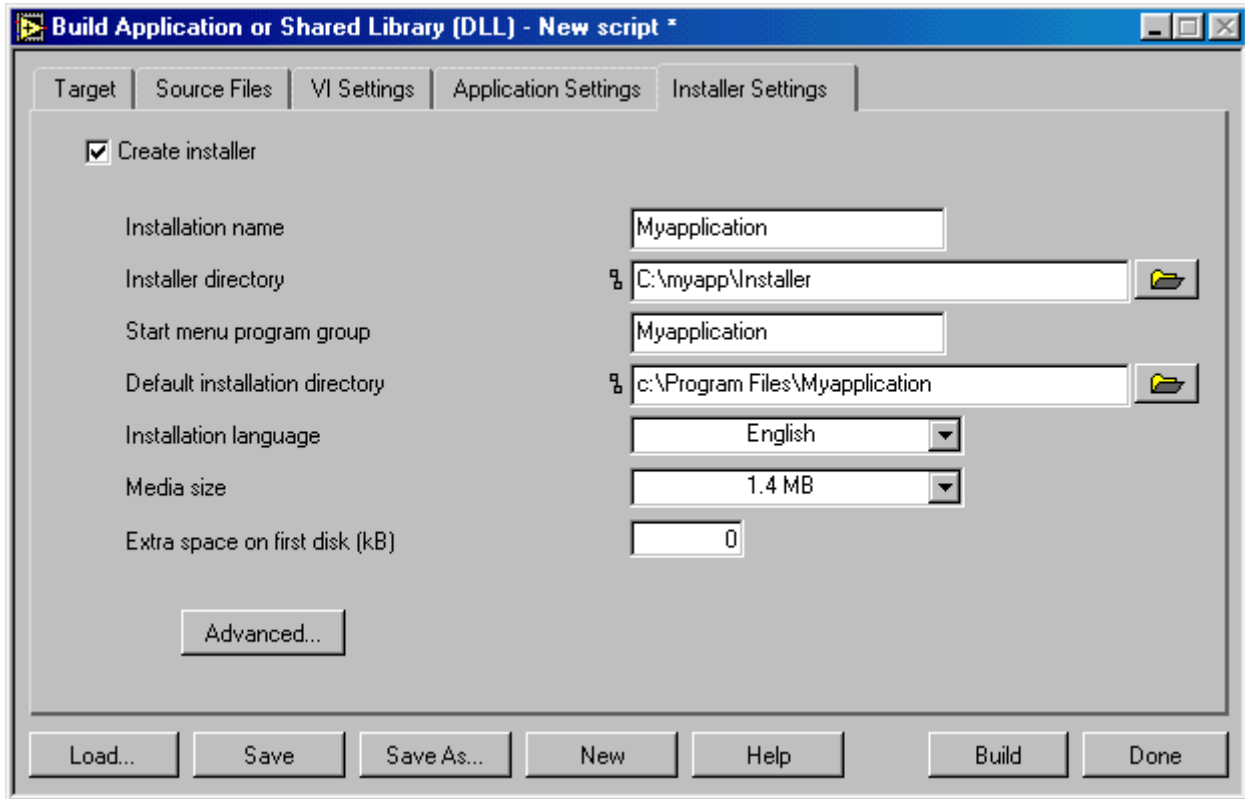


10. Click the **VI Settings** tab. Leave these settings at their default values—the top-level VI runs when opened and the block diagrams and front panels are only saved if they are necessary. Examine these settings; they should resemble the following examples.



11. 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.

12. Click the **Installer Settings** tab. Build a distribution kit for your application that installs into the C:\Program Files\Myapplication directory. Configure the **Installer** tab as shown.



13. Click the **Build** button. The files associated with the installer are then compressed into setup diskettes, which are stored in the C:\myapp\installer\disks directory. A Setup.exe file is created as well, which can be used to install the diskette images. All of these files could be copied to diskettes 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.
14. Select **Done** from the Build Application window to close that utility. It asks you to save a script so you can build this application again. Select **Yes** and name the script myapp.bld. Now 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.
15. 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.

16. Run the Setup.exe file in the C:\myapp\disks\Myapplication directory. You should be guided through a setup process, the executable is created inside the C:\Program Files\Myapplication directory, and you should be able to run the application from **Programs**».

End of Exercise 5-5

Summary, Tips, and Tricks

- 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 a VI's Revision History window at any time by selecting **VI Revision History** from the **Tools** menu.
- The VI Hierarchy window gives you a quick, concise overview of the VIs used in your project. There is also a comparison feature to identify the differences between two VIs.
- LabVIEW features several tools to facilitate larger project development. The Source Code Control Tools contain several utilities for managing LabVIEW code, and the Application Builder enables you to create stand-alone executables or shared libraries (DLLs). Both of these toolkits are included in the LabVIEW Professional Development System.
- 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.

Additional Exercises

- 5-6 Modify Application Exercise VI so that the **Configure Data Acquisition, Acquire Data, Analyze & Present Data**, and **View Analysis File** buttons are disabled and grayed out if the user has not logged in with a valid user name and password.

Hint: Use control references and the Disable Controls VI you built in Exercise 2-8 nodes to modify the buttons.

Save this VI as `Application Exercise (5-6).vi`.

- 5-7 Add one menu button to the menu cluster control in Application Exercise VI so the user can show or hide the LabVIEW Help window. The text of the button should indicate what happens when the user clicks the button, for example, *Show Help* and *Hide Help*. If you notice that the buttons and indicators on the front panel do not have descriptions in the Context Help window, you need to add them in the **Description and Tip** shortcut dialog box for each control or indicator.

Make sure the VI properly tracks the state of the Context Help window. For example, if the user clicks the **Show Help** button to show the Context Help window, but closes the Context Help window from the **Help»Show Context Help** menu, your VI incorrectly assumes that the Context Help window is showing. Save this VI as `Application Exercise (5-7).vi`.



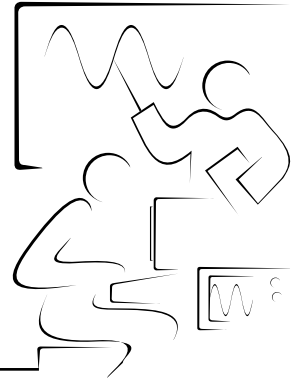
Note Use a Property Node to set the button text. The mechanical action of the button should be Latch When Released. Use a shift register to retain the status of the Context Help window's visibility, and use the Control Help Window function to show and hide the Context Help window.

Notes

Notes

Lesson 6

Performance Issues



This lesson describes how to maximize the performance of your VIs, including how to improve run-time speed and memory use.

You Will Learn:

- A. About LabVIEW multithreading and multitasking.
- B. How to use the **Profile** window.
- C. About methods for speeding up your VIs.
- D. How system memory issues affect LabVIEW performance.
- E. How to optimize memory use and related performance for individual VIs.

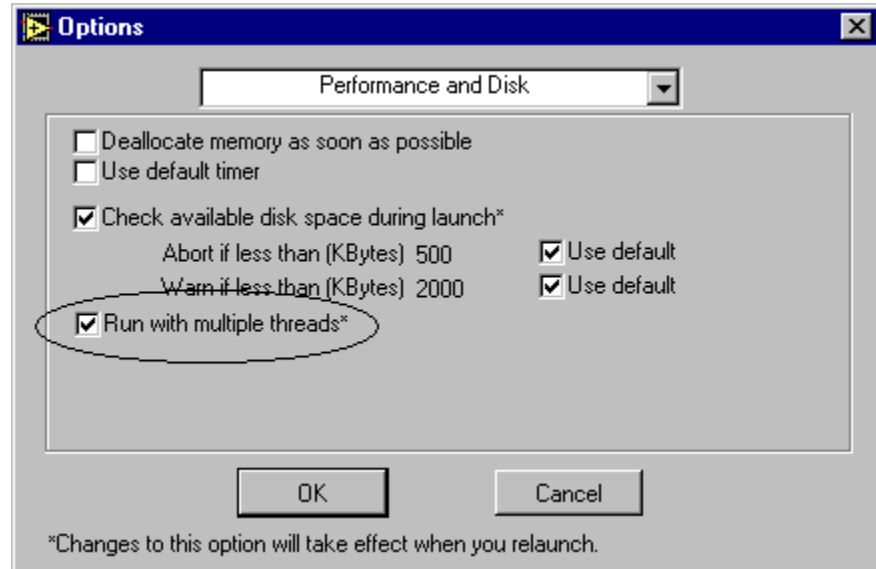
A. LabVIEW Multithreading and Multitasking Overview

In LabVIEW 4 and earlier versions, you could run several VIs simultaneously and the VIs still responded to user input from the cursor or keyboard. To accomplish this, the execution system used cooperative multitasking—each different activity processed one at a time and in turn. Cooperative multitasking works well, but processor time is not evenly distributed to each activity. Executing a long computational routine or displaying a large amount of data on a graph can use a disproportionate amount of the processor's time. The activities take turns with the processor, and a lengthy activity is not divided into smaller, shorter ones. With this architecture, multiple tasks executed under a single thread, or process, in the system.

Versions of LabVIEW after 5.0 support multithreading. With multithreading, different portions of an application can run under different threads, or processes, in a computer system. This architecture allows the operating system to preempt a thread of execution to give processor time to another thread. So, CPU time is more evenly shared among the threads.

To take advantage of multithreading, use LabVIEW 5.0 or later on an operating system that supports multithreading, such as **Windows, Solaris 2, or Concurrent PowerMAX**. On operating systems that do not support multithreading, LabVIEW continues to operate with cooperative multitasking.

By default, LabVIEW takes advantage of multithreading if the operating system supports it. If you want to verify that multithreading is enabled, select **Performance and Disk** from the top pull-down menu in the **Tools»Options** dialog box. You can check or uncheck the box labeled **Run with multiple threads**.



This option is available only on operating systems that support multithreading. If you remove the checkmark from this checkbox, the execution system behaves as though you have only a single thread. Single-thread execution removes some overhead from the execution system. However, you do not benefit from the advantages of multithreading, such as multiple processor support and the ability for a higher priority operation to interrupt a long operation, such as a screen redraw to get better responsiveness from the application. Remove the checkmark from this checkbox for VIs that are not compatible with a multithreaded execution system.

Benefits

One important benefit of multithreaded LabVIEW is the separation of the user interface from block diagram execution. Any activities conducted in the user interface, such as drawing on the front panel, responding to mouse clicks, and so on, operate in their own thread. This prevents the user interface from robbing the block diagram code of execution time. So, displaying a lot of information about a graph does not prevent the block diagram code from executing. Likewise, executing a long computational routine does not prevent the user interface from responding to mouse clicks or keyboard strikes.

Computers with multiple processors benefit even more from multithreading. On a single-processor system, the operating system preempts the threads and distributes time to each thread on the processor. On a multiprocessor computer, threads can run on the multiple processors simultaneously, so more than one activity can occur at the same time.

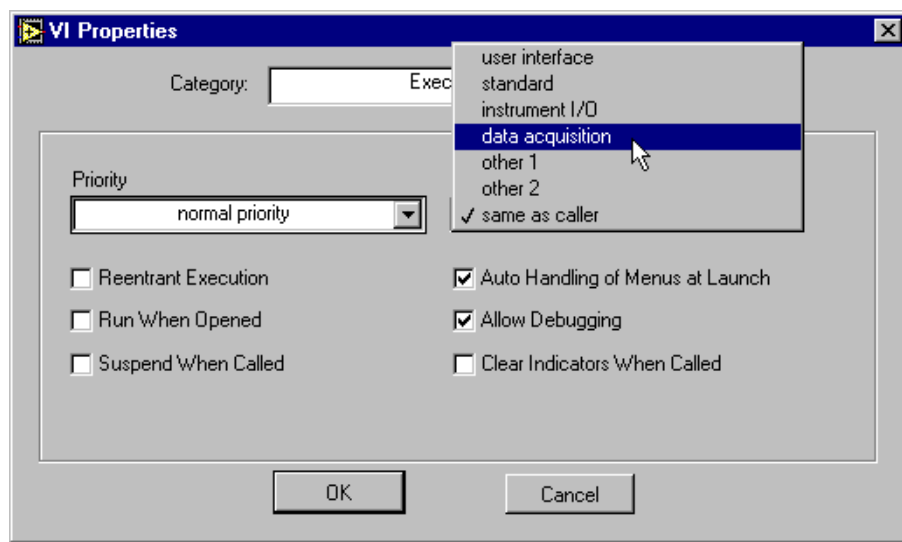
Using Multithreading

When an existing LabVIEW application runs, it takes advantage of the multithreaded system automatically without any modification to the application. However, when working with multiple VIs, there are different classifications of threads, called execution systems, available to organize your application. There are six execution systems available to run VIs:

- User Interface
- Standard
- Instrument I/O
- Data Acquisition
- Other 1
- Other 2

The purpose of having a few different execution systems is to provide some rough partitions for VIs that need to run independently from other VIs. By default, VIs run in the Standard execution system, which runs in separate threads from the user interface. The Instrument I/O execution system is included to prevent VISA, GPIB, and Serial I/O from interfering with other VIs. Similarly, the Data Acquisition execution system is set for the DAQ VIs. While VIs that you write will run correctly if you leave them set to the Standard execution system, you might want to move appropriate VIs to different execution systems. For example, if you are developing instrument drivers, you might want to set your VIs to use the Instrument I/O execution system.

To change the thread category in which a VI runs, select **Execution** from the top pull-down menu in the **VI Properties** dialog box.



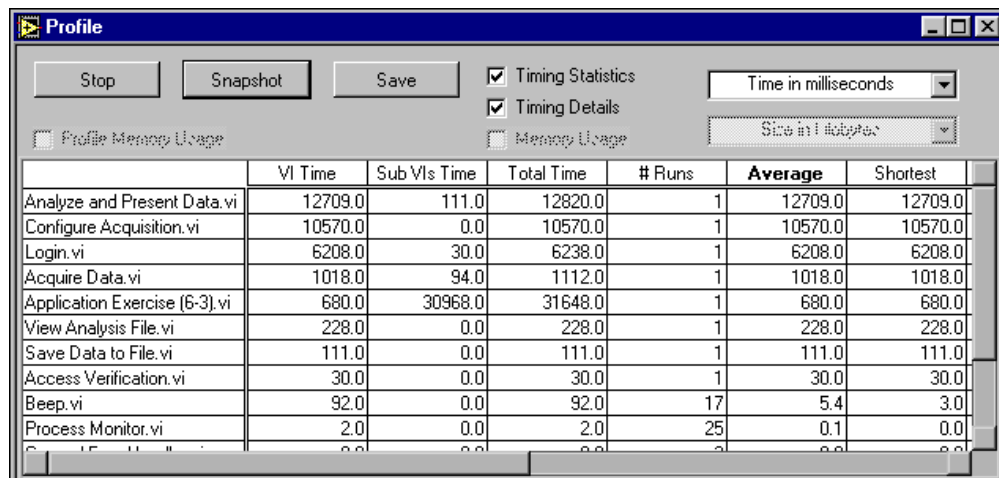
The **Preferred Execution System** list box lists the available categories of execution systems. You also can prioritize parallel tasks in a multithreaded environment by setting the **Priority** of the VI. Within each thread category, you can specify the priority of execution—normal, above normal, high, time critical, subroutine, and background. Normal priority is the same as level 0 priority in previous versions of LabVIEW, above normal priority is the same as level 1 priority, and so on. If there are multiple VIs, the VIs are placed into an execution queue. VIs with higher priority, except the subroutine priority, still execute before lower priority VIs. However, VIs with a subroutine priority level behave differently than VIs with other priority levels. When a VI runs at subroutine priority, it runs in the thread category of its caller, and no other VI can run in that thread until that VI or its subVIs complete. Subroutine priority VIs can call other subroutine priority VIs only. Use subroutine priority VIs only when you want to run a simple computation with no interactive elements. You can skip the execution of a subroutine priority subVI when it is busy by right-clicking the subVI and selecting **Skip Subroutine Call if Busy**. Use this option when you are calling a shared subroutine from a high-priority VI but you do not want to wait for the subroutine VI to become available.

Exercise caution when setting the priority levels for VIs. Using priorities to control execution order might not produce the results you expect. For example, if you use the priority setting incorrectly, lower priority tasks might never execute. As you will see later in this lesson, strategic use of Wait functions within VIs also can be a very effective way of optimizing your LabVIEW code. Refer to the *LabVIEW Help* for more information about how to use Wait functions.

B. The Profile Window

The **Profile** window is a powerful tool for analyzing how your application uses execution time as well as memory. With this information, you can identify the specific VIs or parts of VIs you need to optimize. For example, if you notice that a particular subVI spends a great deal of time updating the display, you can focus your attention on improving the display performance of that VI.

The **Profile** window displays the performance information for all VIs in memory in an interactive tabular format. From the window, you can choose the type of information to gather and sort the information by category. You also can monitor subVI performance within different VIs. To show the **Profile** window, select **Tools»Advanced»Profile VIs**. The following window appears.



	VI Time	Sub VIs Time	Total Time	# Runs	Average	Shortest
Analyze and Present Data.vi	12709.0	111.0	12820.0	1	12709.0	12709.0
Configure Acquisition.vi	10570.0	0.0	10570.0	1	10570.0	10570.0
Login.vi	6208.0	30.0	6238.0	1	6208.0	6208.0
Acquire Data.vi	1018.0	94.0	1112.0	1	1018.0	1018.0
Application Exercise (6-3).vi	680.0	30968.0	31648.0	1	680.0	680.0
View Analysis File.vi	228.0	0.0	228.0	1	228.0	228.0
Save Data to File.vi	111.0	0.0	111.0	1	111.0	111.0
Access Verification.vi	30.0	0.0	30.0	1	30.0	30.0
Beep.vi	92.0	0.0	92.0	17	5.4	3.0
Process Monitor.vi	2.0	0.0	2.0	25	0.1	0.0

Notice that you must select the **Profile Memory Usage** option before starting a profiling session. Collecting information about VI memory use adds a significant amount of overhead to VI execution, which affects the accuracy of any timing statistics you gather during the profiling session. Therefore, you should perform memory profiling separate from time profiling.

Many of the options in the **Profile** window are available only after you begin a profiling session. During a profiling session, you can grab a snapshot of the available data and save the data to an ASCII spreadsheet file. The timing measurements accumulate each time you run a VI.



Note All statistics measured in a profiling session are collected for a complete run of a VI, not a partial run of a VI.

Execution Time Statistics

To collect execution time statistics with the **Profile** window, click the **Start** button. The previous example shows the **Profile** window after a time profiling session.

The VI Time statistic column shows the amount of time spent executing the VI and displaying its data, as well as the time spent by the user interacting with the front panel. This time consists of five subcategories, which you can show by selecting the **Timing Details** option. The subcategories are Diagram, Display, Draw, Tracking, and Locals. Refer to the *LabVIEW Help* or the *LabVIEW User Manual* for more information about these statistics.

The SubVIs statistic column shows the total execution time for all the subVIs called by the main VI. In addition, timing information can be displayed in seconds, milliseconds, or microseconds.

The Total Time statistic column shows the sum of the VI and subVIs values, which represents the total execution time for the main VI.

If you select the **Timing Statistics** option, four new categories of information appear. **# Runs** displays how many times each VI has been executed. The **Average** value represents the VI time value divided by the **# Runs**, or the average amount of time the VI takes to run. **Shortest** and **Longest** show the least and greatest amount of time required for a run of the VI.

Memory Statistics

To collect memory statistics with the **Profile** window, select the **Profile Memory Usage** option before starting the profiling session. You also can select the **Profile Memory Usage** option after starting the Profiler to collect additional memory information.

The **Profile** window displays two sets of memory use data. One set of data shows the number of bytes of memory used, and the other shows the blocks of memory. LabVIEW stores data such as arrays, strings, and paths in contiguous blocks of memory. If a VI uses a large number of blocks of memory, the memory can fragment, which degrades LabVIEW performance in general, not VI execution.

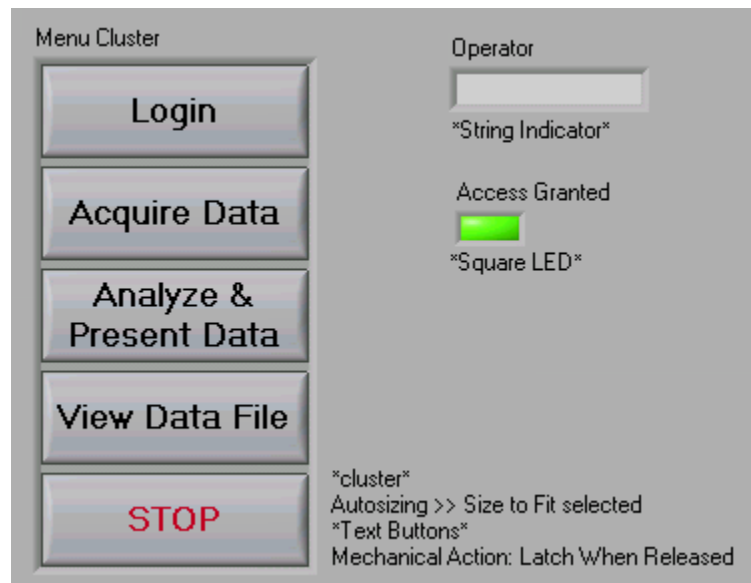
The Average Bytes statistic column shows the average number of bytes of memory used by a VI's data space during its execution. **Min Bytes** and **Max Bytes** represent the least and greatest amount of memory used by a VI during an individual run. **Average Blocks** indicates how many blocks of memory a VI needs on average, while the **Min Blocks** and **Max Blocks** show the fewest and greatest number of blocks of memory used by a VI during an individual run.

Exercise 6-1 The Profile Window

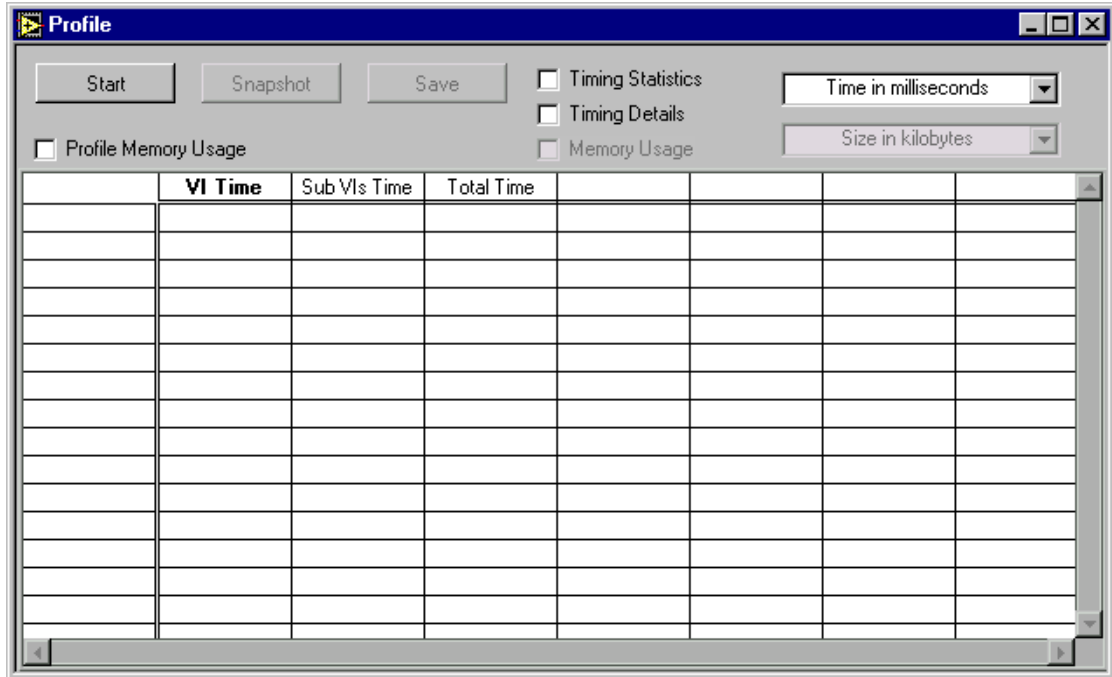
Objective: To use the Profile window to get information about how your application VI runs.

The **Profile** window is a powerful tool for analyzing how your application uses execution time as well as memory. With this information, you can identify the areas of your VI that you need to optimize. In this exercise, examine the project just completed using the **Profile** window.

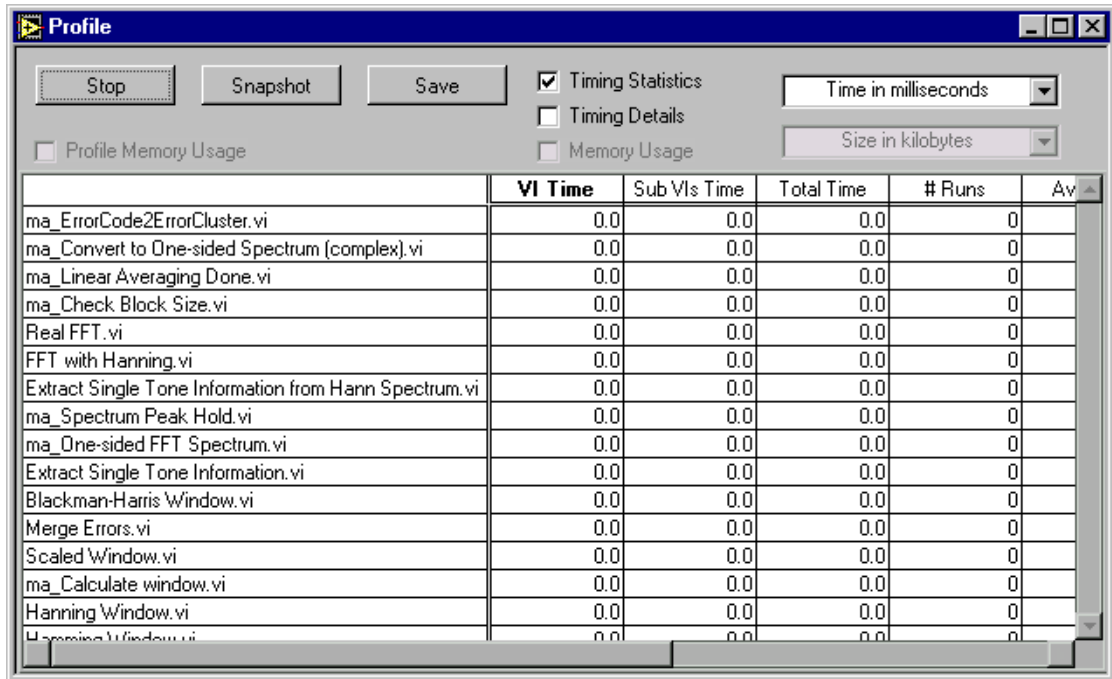
1. Open your final project, called CD Application Exercise VI. If you did not complete this final project, you can open the solution.



2. Before running your final project, select **Tools»Advanced»Profile VIs** to display the **Profile** window. The **Profile** window is not yet capturing data.



3. To collect timing information about your final project, place a checkmark in the **Timing Statistics** checkbox and click the **Start** button.

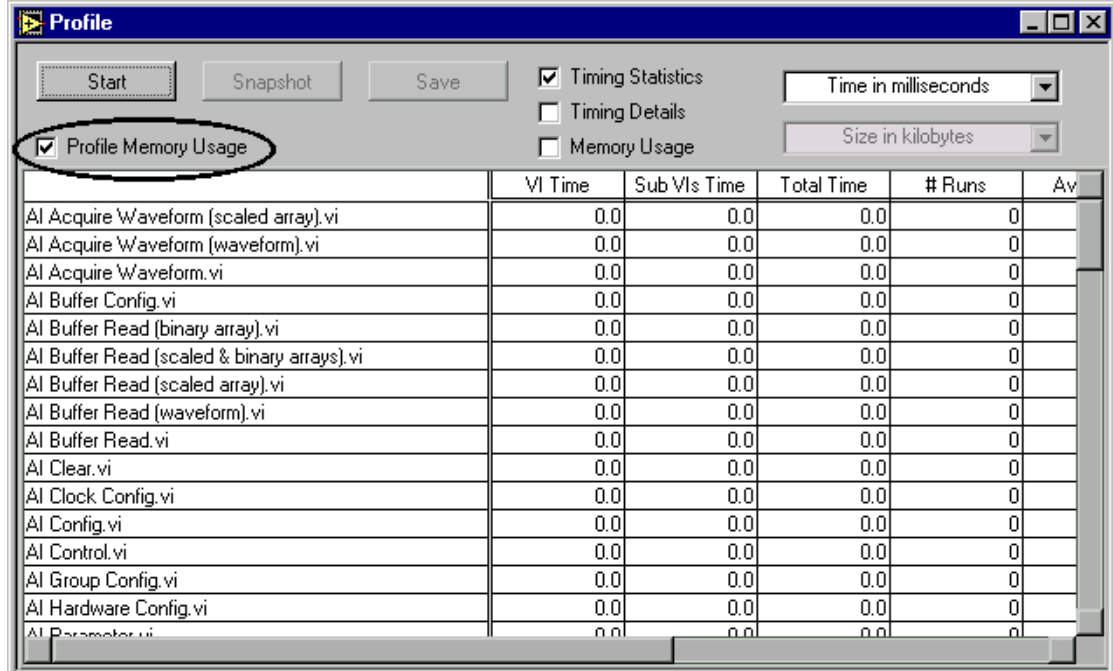


You are now capturing timing information about the VIs in memory.

4. Run the CD Application Exercise VI and go through the steps as if you were a typical user. At any time you can return to the **Profile** window and take a snapshot of timing. Taking a snapshot does not effect the final timing statistics.
5. When you finish using your project, return to the **Profile** window and click the **Snapshot** button again. You can sort any column of data by clicking the column heading. You also can select whether the timing statistics are displayed in seconds, milliseconds, or microseconds. By double-clicking VIs listed in the **Profile** window, you can view the statistics of the VI that compose it. This allows you to drill down to the subVIs to see where time is being spent. You can display the front panel of the individual VIs by right-clicking the VI listed in the **Profile** window.

Sort the Total Time column and look at where the CD Application Exercise VI spends most of its time. Notice how these VIs correspond with the VIs that require user interaction. Waiting for user input is often the portion of execution which requires the most time.

In the **Profile** window you can capture timing statistics and information about VI memory usage. To capture memory usage, place a checkmark in the **Profile Memory Usage** checkbox.



The screenshot shows the LabVIEW Profile window. At the top, there are buttons for 'Start', 'Snapshot', and 'Save'. To the right, there are checkboxes for 'Timing Statistics' (checked), 'Timing Details', and 'Memory Usage'. Below these are dropdown menus for 'Time in milliseconds' and 'Size in kilobytes'. A checkbox for 'Profile Memory Usage' is circled in red. Below the controls is a table with the following columns: VI Name, VI Time, Sub VIs Time, Total Time, # Runs, and Av. The table lists various VIs, all with 0.0 values for time and 0 for runs.

	VI Time	Sub VIs Time	Total Time	# Runs	Av
AI Acquire Waveform (scaled array).vi	0.0	0.0	0.0	0	
AI Acquire Waveform (waveform).vi	0.0	0.0	0.0	0	
AI Acquire Waveform.vi	0.0	0.0	0.0	0	
AI Buffer Config.vi	0.0	0.0	0.0	0	
AI Buffer Read (binary array).vi	0.0	0.0	0.0	0	
AI Buffer Read (scaled & binary arrays).vi	0.0	0.0	0.0	0	
AI Buffer Read (scaled array).vi	0.0	0.0	0.0	0	
AI Buffer Read (waveform).vi	0.0	0.0	0.0	0	
AI Buffer Read.vi	0.0	0.0	0.0	0	
AI Clear.vi	0.0	0.0	0.0	0	
AI Clock Config.vi	0.0	0.0	0.0	0	
AI Config.vi	0.0	0.0	0.0	0	
AI Control.vi	0.0	0.0	0.0	0	
AI Group Config.vi	0.0	0.0	0.0	0	
AI Hardware Config.vi	0.0	0.0	0.0	0	
AI Parameter.vi	0.0	0.0	0.0	0	



Note Collecting information about VI memory usage requires overhead which can create misleading timing statistics.

6. Gather memory usage statistics about the CD Application Exercise VI.

Where is memory used and why?

Notice which VIs fragment memory, and look to see which VIs use many blocks. Why do you suppose these VIs use blocks? Having fragmented memory causes VIs to execute more slowly because moving data around in memory takes overhead.

End of Exercise 6-1

C. Speeding Up Your VIs

LabVIEW compiles your VIs and produces code that generally executes very quickly. However, when working on time-critical applications, you need to use programming techniques to obtain the best possible execution speed. Consider the following three areas when speeding up your VIs:

- Input/Output—files, instrument control, data acquisition, and networking
- Screen display—efficient controls and displays
- Memory management—efficient use of arrays, strings, and data structures

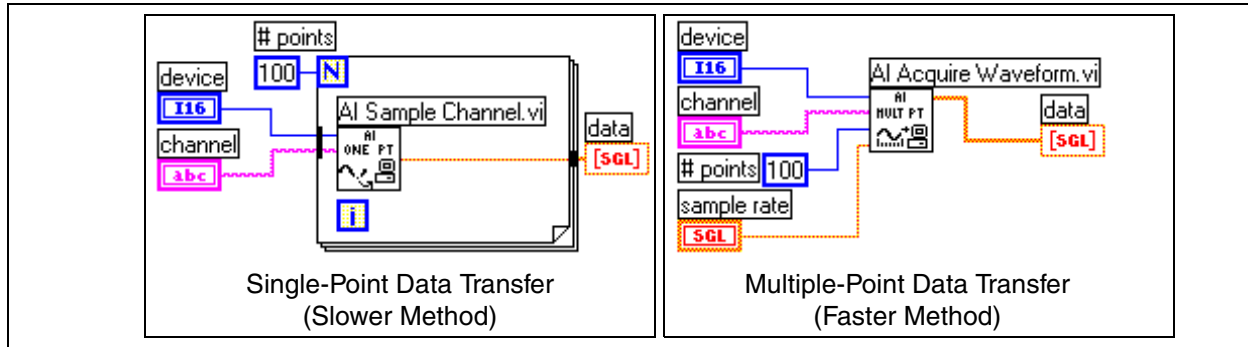
Other factors, such as execution overhead and subVI call overhead, usually have minimal effects on execution speed.

Input/Output

Input/Output (I/O) calls generally take much more time than a computational operation. For example, a simple serial port read operation can have an associated overhead of several milliseconds. This overhead is present not only in LabVIEW, but also in other applications. The reason for this overhead is that an I/O call involves transferring information through several layers of an operating system.

The best method of reducing this I/O overhead is to minimize the number of I/O calls you make in your application. Structure your application so that you transfer larger amounts of data with each call, instead of making several I/O calls that transfer a small amount of data.

For example, consider creating a data acquisition (DAQ) application that acquires 100 points of data. For faster execution, use a multi-point data transfer function such as the AI Acquire Waveform VI, instead of using a single-point data transfer function such as the AI Sample Channel VI. To acquire 100 points, use the AI Acquire Waveform VI with an input specifying that you want 100 points. This is faster than using the AI Sample Channel VI in a loop with a wait function to establish the timing.



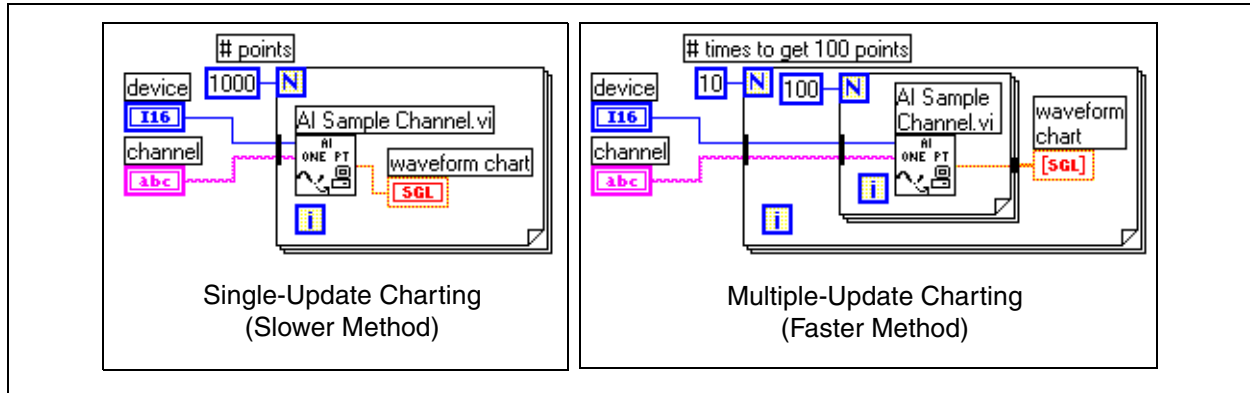
In the previous example, overhead for the AI Acquire Waveform VI is roughly the same as the overhead for a single call to the AI Sample Channel VI, even though it is transferring much more data. In addition, the data collected by the AI Acquire Waveform VI uses hardware timers to control the sampling. Calling AI Sample Channel in a loop does not provide data collected at a constant sample rate.

Screen Display

Updating controls on a front panel is another time-consuming task in an application. While multithreading helps to reduce the effect that display updates have on overall execution time, complicated displays, such as graphs and charts, can adversely affect execution speed. This effect can become significant on the LabVIEW platforms that do not support multithreading. Although most indicators do not redraw when they receive data values that are the same as the old data, graphs and charts always redraw. To minimize this overhead, keep front panel displays simple, and try to reduce the number of front panel objects. Disabling autoscaling, scale markers, and grids on graphs and charts improves their efficiency.

The design of subVIs also can reduce display overhead. If subVIs have front panels that remain closed during execution, none of the controls on the front panel can affect the overall VI execution speed.

As shown in the following block diagram, you can reduce display overhead by minimizing the number of times your VI updates the display. Drawing data on the screen is an I/O operation, similar to accessing a file or GPIB instrument. For example, you can update a waveform chart one point at a time, or several points at a time. You get much higher data display rates if you collect your chart data into arrays so that you can display several points at a time. This way, you reduce the amount of I/O overhead required to update the indicator.



Other Issues

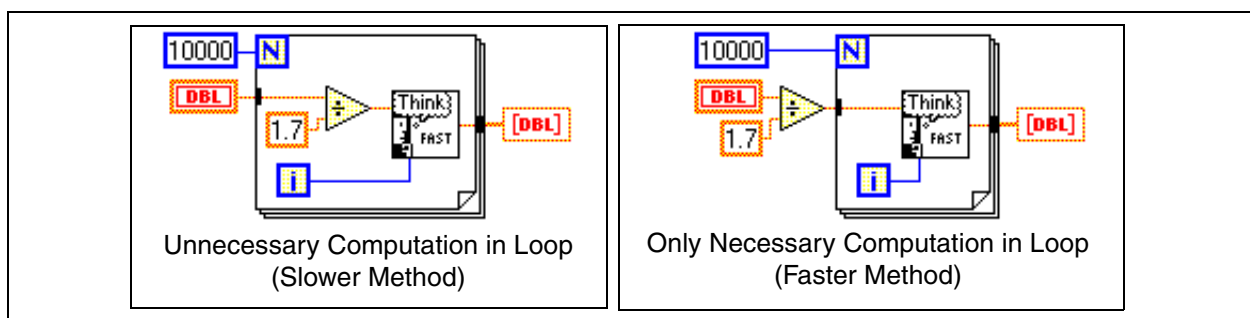
SubVI Overhead

Each call to a subVI involves a certain amount of overhead. This overhead is fairly small, on the order of tens of microseconds, especially in comparison to I/O overhead and display overhead, which can range from milliseconds to tens of milliseconds. However, if you make 10,000 calls to a subVI in a loop, the overhead could significantly affect your execution speed. In this case, you might consider embedding the loop in the subVI.

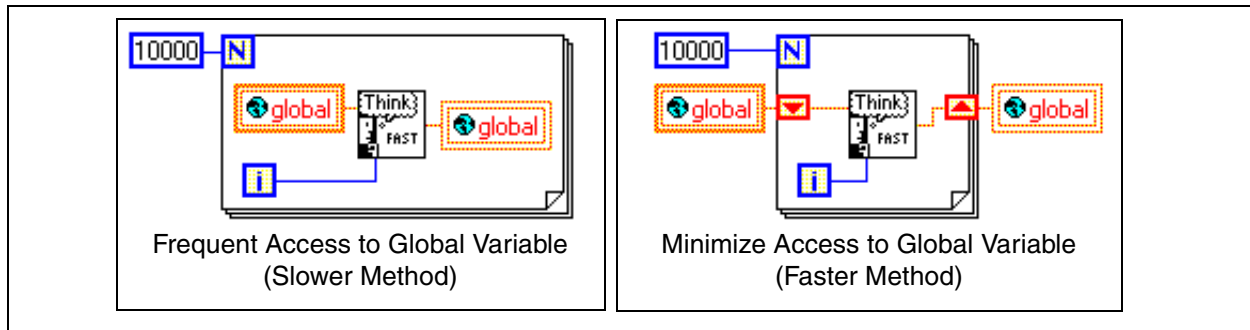
Another way to minimize subVI overhead is to turn your subVIs into subroutines by selecting **Execution** from the top pull-down menu in the **File»VI Properties** dialog box. However, there are some trade-offs. Subroutines cannot display front panel data, call timing or dialog box functions, or multitask with other VIs. Subroutines are generally most appropriate for VIs that do not require user interaction and are short, frequently executed tasks.

Unnecessary Computation in Loops

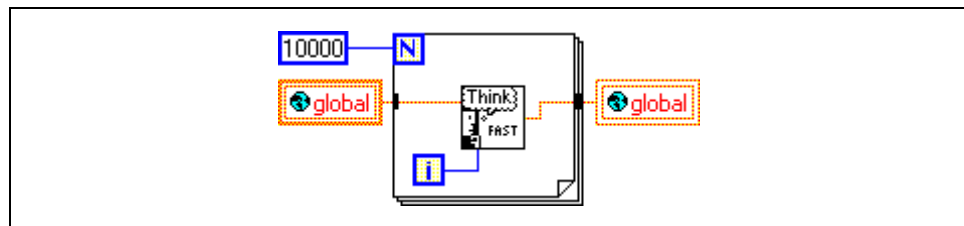
Avoid placing calculations in loops if the calculation produces the same value for every iteration. Instead, move the calculation out of the loop and pass the result into the loop. For example, consider the following two block diagrams. The result of the division is the same every time through the loop, so you can increase performance by moving the division out of the loop.



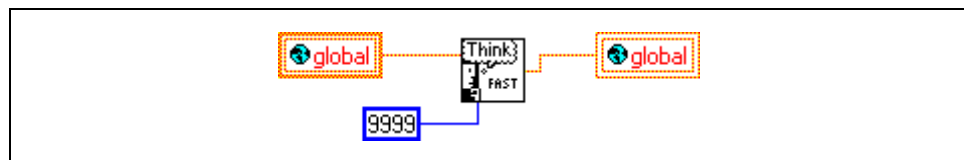
Also, avoid accessing local and global variables unnecessarily in loops. For example, the following first block diagram wastes time by reading from the global variable and writing to the global variable during each iteration of the loop. If you know that the global variable is not read from or written to by another block diagram during the loop, consider using shift registers to store the data, as shown in the second block diagram.



Notice that you need the shift registers to pass the new value from the subVI to the next iteration of the loop. Beginning LabVIEW users commonly omit these shift registers. Without using a shift register, the results from the subVI are never passed back to the subVI as the new input value, as shown in the following block diagram.

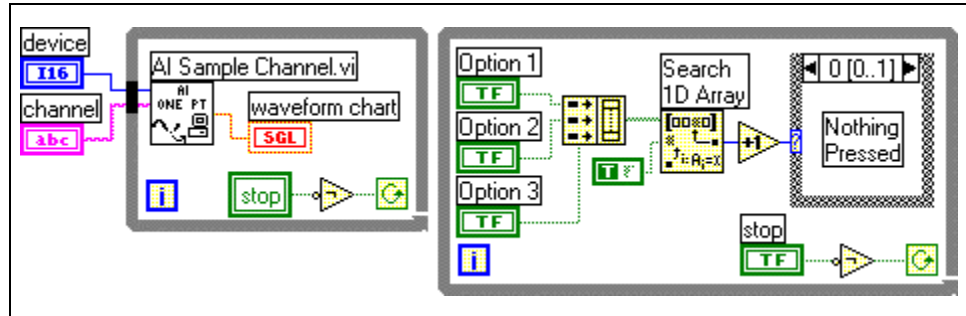


In the previous block diagram, the global variable is read once before the loop begins, and the same value is passed to the function 10,000 times. The result of the loop is the same as if you had written the code as shown in the following block diagram.

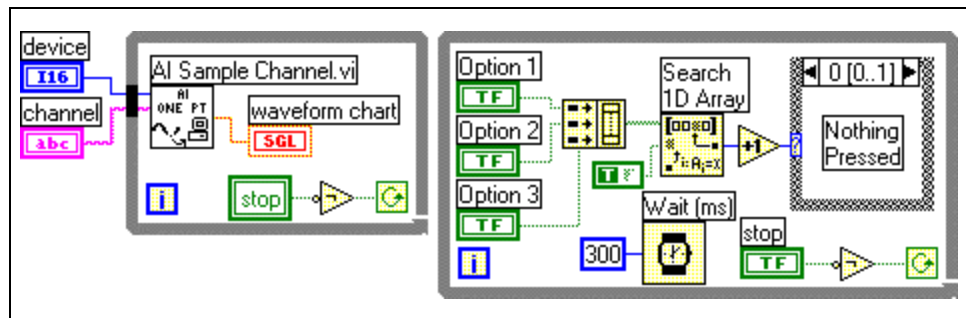


Parallel Diagrams

When several loops run in parallel on a block diagram, LabVIEW changes between the loops periodically. If some of these loops are less important than others, you should use the Wait (ms) function to ensure that the less important loops use less time. For example, consider the following block diagram.



The two loops run in parallel. One of the loops acquires data and must execute as frequently as possible. The other loop monitors user input. Currently, both loops get equal time. The loop monitoring the user input can execute several hundred times per second. In reality, this loop needs to execute only a few times per second, because the user cannot make changes to the interface very quickly. As shown in the following block diagram, you can call the Wait (ms) function in the user interface loop to give significantly more execution time to the other loop.

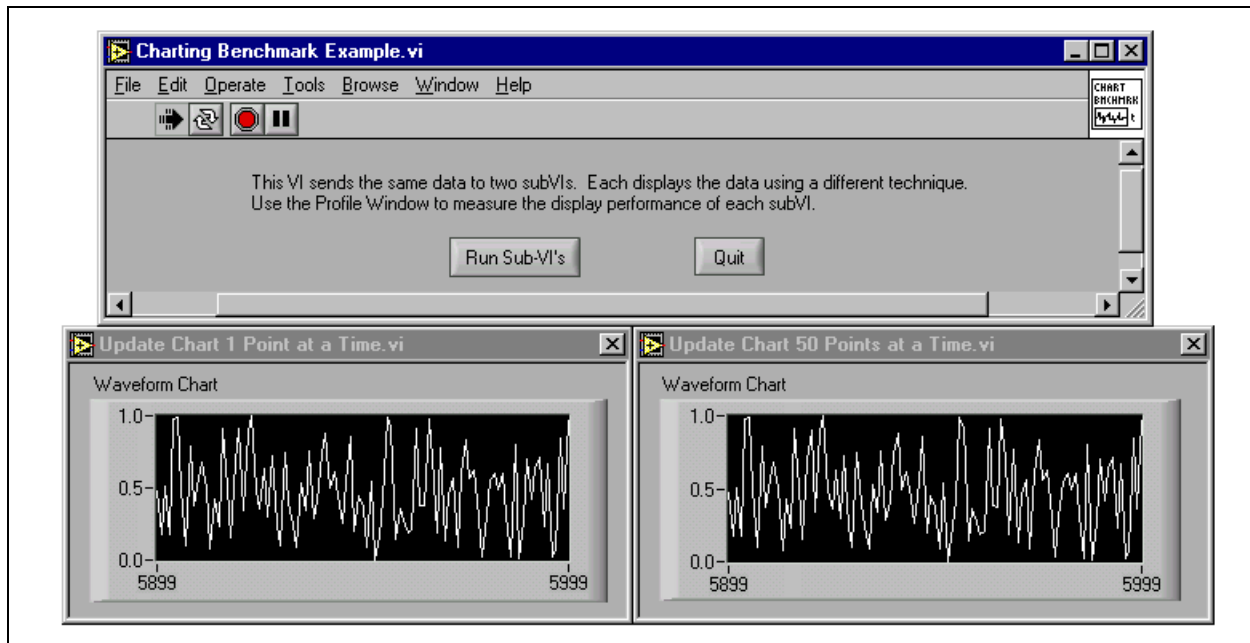


Exercise 6-2 Charting Benchmark Example VI

Objective: To observe the relative speeds of single-point and multiple-point charting using the Profile window and the effects of multithreading on systems that support this feature.

When you open the Charting Benchmark Example VI, two other subVI front panels also open. These two subVIs plot the same data under different conditions.

Front Panels



1. If you are running LabVIEW on an operating system that supports multithreading (**Windows, Solaris2, or Concurrent PowerMAX**), complete steps a through c. If you are on a system that does not support multithreading, skip to step 2.
 - a. Configure LabVIEW so that it does not use multithreading when it runs VIs. Select **Tools»Options** to display the **Options** dialog box.
 - b. From the top pull-down menu of the **Options** dialog box, select **Performance and Disk**. In the Performance and Disk section, remove the checkmark from the **Run with multiple threads** checkbox. Click **OK**.
 - c. Exit and restart LabVIEW so that the updated preferences take effect. LabVIEW is now configured to run VIs under a single thread, using co-operative multitasking.

2. Close all other VIs you might have open.
3. Open the Charting Benchmark Example VI. The VI is already built for you. When you open it, two other VIs also open.

The Charting Benchmark Example VI generates an array of 1,000 random numbers. When you click the **Run SubVIs** button, two subVIs are executed. The Update Chart 1 Point at a Time subVI updates a chart 1,000 times. The second subVI, Update Chart 50 Points at a Time, updates a chart 20 times, displaying 50 points each time. Select **Tools»Advanced»Profile VIs**.

	VI Time	Sub VIs Time	Total Time			
Update Chart 50 Points at a Time.vi	0.0	0.0	0.0			
Update Chart 1 Point at a Time.vi	0.0	0.0	0.0			
Charting Benchmark Example.vi	0.0	0.0	0.0			

4. Move the window so that it does not overlap any of the VI front panels.
5. Click the **Start** button in the **Profile** window. You see the VI Time, SubVIs Time, and Total Time statistics in the table.
6. Run the Charting Benchmark Example VI. The VI does nothing until you click the **Run SubVIs** button. Run the subVIs at least three or four times. Then click the **Quit** button to stop the Charting Benchmark Example VI.
7. After running the subVIs several times, click the **Snapshot** button in the **Profile** window. Locate Charting Benchmark Example VI in the **Profile** window and double-click it to display its subVIs. Notice the VI execution times for the two subVIs. Recall that these are cumulative times, not the amount of time needed to execute the subVIs once. Notice that updating the chart one point at a time is the slowest method, and that updating the chart several points at a time is the fastest method.

8. Select the **Timing Statistics** option to view the average execution time for the subVIs. Then select the **Timing Details** option and examine the Display and Draw statistics. Notice that the Display statistic is largest for the Update Chart 1 Point at a Time subVI and is very low for the Update Chart 50 Points at a Time VI. Record the average execution time in the following table.

Refer to the *LabVIEW Help* or the *LabVIEW User Manual* for more information about these statistics.

9. Click the **Stop** button in the **Profile** window to stop the profiling session. Then close the **Profile** window.
10. Close all open VIs and do not save any changes.
11. If you are running LabVIEW on a multithreading operating system, place a checkmark in the **Run with multiple threads** checkbox on the **Execution** page in the **File>VI Properties** dialog box and restart LabVIEW. Repeat steps 3 through 10, noting in the table how much faster the VIs execute when the display is running under its own execution subsystem.

Table 6-1. Record of Execution Times

	Average Execution Time		
	Multithreading Off	Multithreading On	Percent Faster
Charting Benchmark Example VI			%
Update Chart 1 Point at a Time VI			%
Update Chart 50 Points at a Time VI			%

End of Exercise 6-2

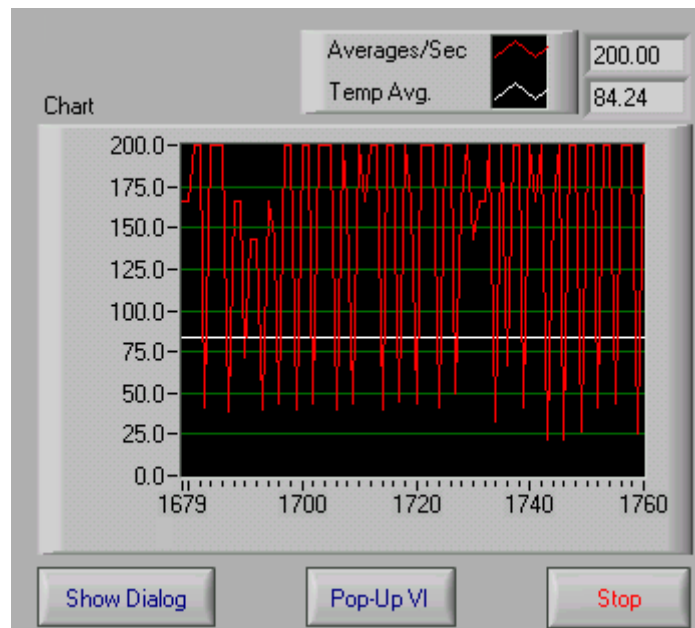
Exercise 6-3 Dialog & SubVI Demo VI

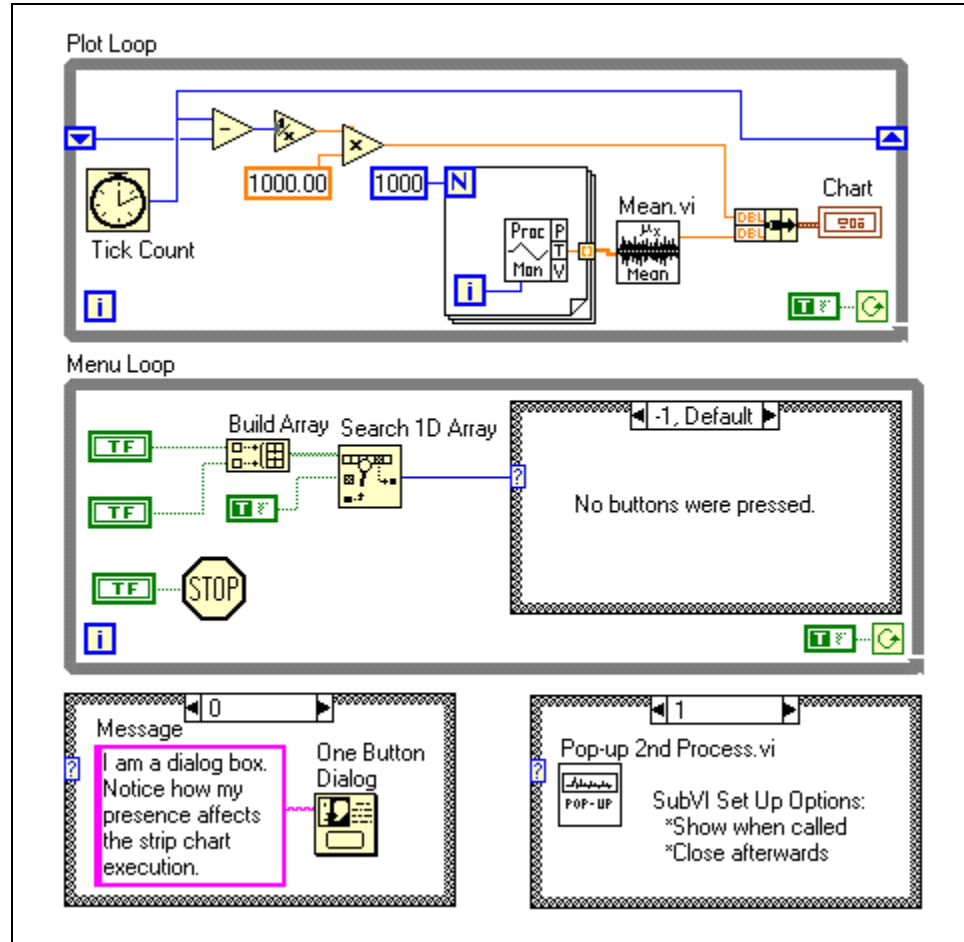
Objective: To observe how dataflow structure can affect execution speed.

Load and run a VI that plots a waveform pattern. Notice how the dataflow affects the execution of the VI.

Front Panel

1. If you are running LabVIEW on an operating system that supports multithreading (**Windows, Solaris2, or Concurrent PowerMAX**), complete steps a through c. If you are on a system that does not support multithreading, skip to step 2.
 - a. Configure LabVIEW so that it does not use multithreading when it runs VIs. Select **Tools»Options** to display the **Options** dialog box.
 - b. From the top pull-down menu of the **Options** dialog box, select **Performance and Disk**. In the Performance and Disk section, remove the checkmark from the **Run with multiple threads** checkbox. Click **OK**.
 - c. Exit and restart LabVIEW so that the updated preferences take effect. LabVIEW is now configured to run VIs under a single thread, using co-operative multitasking.
2. Open the Dialog & SubVI Demo VI. The VI is already built for you. The front panel contains a strip chart and several option buttons. This chart shows two pieces of data, the running average of a simulated temperature and a red plot showing how quickly these averages are calculated.





- Open the block diagram and examine it. Two While Loops run in parallel. One loop handles data collection and analysis while the other handles the user interface.
- Run the VI. Notice the plot speed and the averages/second being calculated.

Notice that the Averages/Sec value is periodic and dips very low. The block diagram contains two While Loops running in parallel. The Plot Loop (top) is the one generating the temperature average and the number of averages per second. The dips in performance happen when the Menu Loop executes and checks the values of the buttons on the front panel. Performance increases when the Plot Loop runs.

- Click the **Show Dialog** button. Notice the effect on the plot speed and the Averages/Sec value. Recall that a loop cannot begin its next iteration until the entire block diagram inside it finishes executing. In the Menu Loop, the values from the buttons are read to determine which case should be executed. If you click the **Show Dialog** button, Case 0 is executed. The case does not complete until you click the **OK** button in

the dialog box. Thus, while the dialog box is displayed, the Plot Loop does not need to share the processor with the Menu Loop.

6. Click the **Pop-Up VI** button. The Pop-up 2nd Process VI opens its front panel and runs. Notice the effect on the first VI's Averages/Sec value. As in the situation when you clicked the **Show Dialog** button, the Menu Loop no longer executes until the Pop-up 2nd Process VI ends. However, LabVIEW runs the second VI at the same time as the original VI. The dips in performance appear when the second VI runs. Therefore, LabVIEW executes multiple VIs in the same way it executes parallel loops—each process gets an equal time-slice.
7. Click the **Stop** button.
8. Modify the Menu Loop so it includes a Wait (ms) function. Wire a constant value of 300 to the input of the function. Notice how the speed of the Plot Loop changes.
9. Close the VI and do not save any changes.
10. If you are running LabVIEW on a multithreaded operating system, place a checkmark in the **Run with Multiple Threads** checkbox on the **Execution** page in the **File»VI Properties** dialog box. Exit and restart LabVIEW. Repeat steps 2 through 9 and notice how multithreading affects the execution speed of the Dialog & SubVI Demo VI.

If you want to magnify the results of this exercise, move windows with your cursor while the Dialog & SubVI Demo VI runs. This increases the processor workload creating more drastic results.

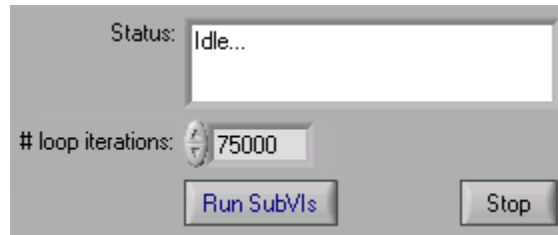
End of Exercise 6-3

Exercise 6-4 Computation & Global Benchmark VI

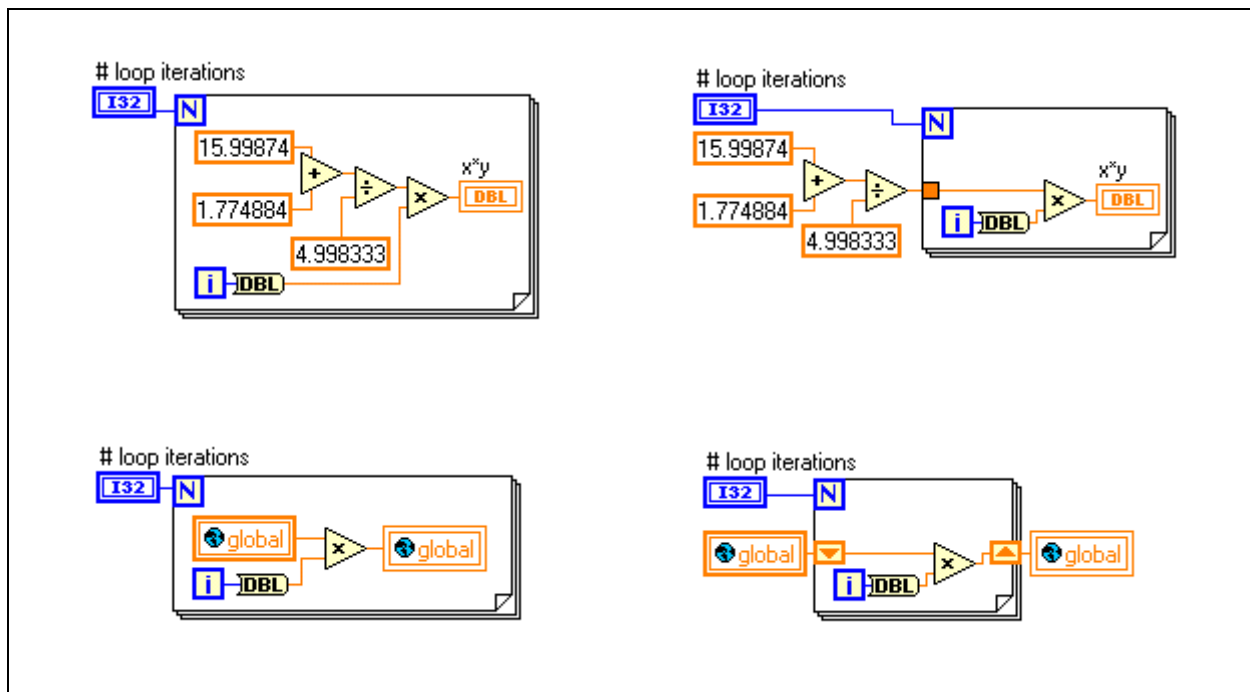
Objective: To observe the effects of unnecessary computation and global variable activity on VI execution speed.

Use the **Profile** window to compare the performance of VIs that perform the same tasks using different programming techniques.

Front Panel



1. Close all other VIs you might have open.
2. Open the Computation & Global Benchmark VI. The VI is already built for you.
3. Select **Tools»Advanced»Profile VIs**. Click the **Start** button in the **Profile** window. Do not gather any memory statistics.
4. Run the Computation & Global Benchmark VI. Click the **Run SubVIs** button to run the following four subVIs. Run the subVIs at least three or four times.



5. After running the subVIs several times, stop the VI and click the **Snapshot** button in the **Profile** window. Locate the Computation and Global Benchmark VI in the **Profile** window and double-click it to display its subVIs if they are not already displayed.

Notice the amount of time spent in each subVI. Specifically, compare the amount of time spent executing Update Globals Inside Loop VI versus Globals Outside of Loop VI and the time spent executing Unnecessary Loop Computations VI versus Unnecessary Computations Removed VI. Is this the behavior you expected? How would you apply what you have learned in this exercise to your programming?

6. Click the **Stop** button in the **Profile** window to stop the profiling session. Then close the **Profile** window.
7. Close the VI. Do not save any changes.

End of Exercise 6-4

D. System Memory Issues

LabVIEW transparently handles many of the details that you normally deal with in text-based programming languages. One of the main challenges of programming with a text-based language is memory use. In a text-based programming language, you must allocate memory before you use it and deallocate it when you are finished. You also must be particularly careful not to accidentally write past the end of the memory you have allocated. Failure to allocate memory or to allocate enough memory is one of the biggest mistakes that programmers make in text-based programming languages.

LabVIEW's dataflow programming removes much of the difficulty of managing memory. In LabVIEW, you do not allocate variables, nor assign values to and read values from them. Instead, you create a block diagram with graphical connections representing the transition of data.

Functions that generate data allocate storage for those data. When data are no longer needed, LabVIEW deallocates the associated memory. When you add new information to an array or a string, LabVIEW allocates the necessary memory.

This automatic memory handling is one of the chief benefits of LabVIEW. However, because it is automatic, you have less control over it. You should understand how LabVIEW allocates and deallocates memory so you can design applications with smaller memory requirements. Also, an understanding of how to minimize memory use can help you increase VI execution speed, because memory allocation and copying data can take a considerable amount of time.

LabVIEW Memory

Windows, Sun, and HP-UX—LabVIEW allocates memory dynamically, taking as much as needed. This process is transparent to the user.

Macintosh—LabVIEW allocates a single block of memory at launch time, out of which all subsequent allocations are performed. When you load a VI, its components are loaded into this block of memory. Likewise, when you run a VI, all the memory that it manipulates is allocated from this block.

On the Macintosh, you configure the amount of memory that LabVIEW allocates at launch time by selecting the **Get Info** option from the **File** menu in the Finder. Note that if LabVIEW runs out of memory, it cannot increase the size of this memory pool. Therefore, you should set this parameter as large as possible. Make sure to leave enough memory for any other applications that you want to run at the same time as LabVIEW.

Virtual Memory

When using Windows or Macintosh, you can use virtual memory to increase the amount of memory available for your applications. Virtual memory uses available disk space for RAM storage. If you allocate a large amount of virtual memory, applications perceive this as memory that is available for storage.

On the Macintosh, you allocate virtual memory using the Memory device in the Control Panel folder. **Windows, Sun, and HP-UX** automatically manage virtual memory allocation.

LabVIEW does not differentiate between RAM and virtual memory. The operating system hides the fact that the memory is virtual. However, accessing data stored in virtual memory is much slower than accessing data stored in physical RAM. With virtual memory, you might occasionally notice more sluggish performance because data are swapped to and from the hard disk by the operating system. Virtual memory can help run larger applications, but it is probably not appropriate for applications that have critical time constraints.

VI Components

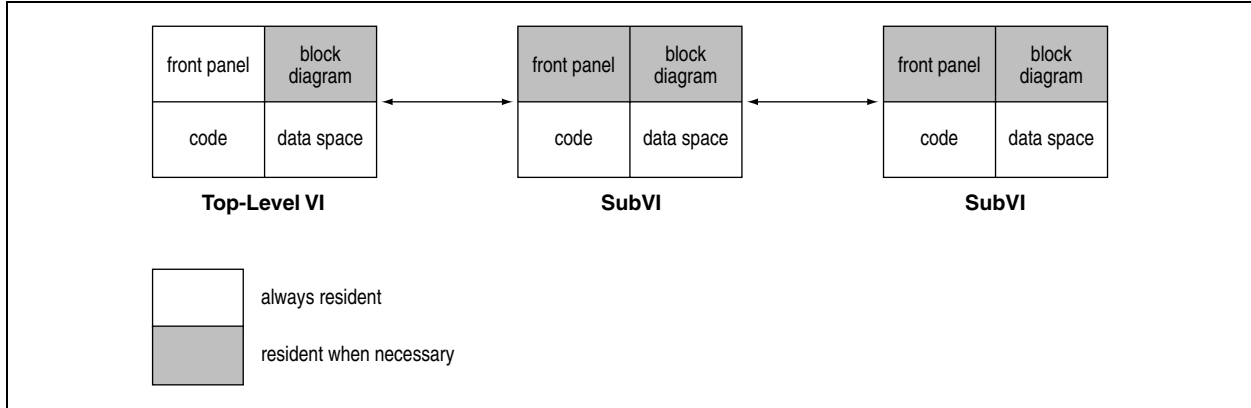
VIs have the following major components:

- Front Panel
- Block Diagram
- Code—block diagram compiled to machine code
- Data—control and indicator values, default data, block diagram constant data, and so on

When you load a VI, LabVIEW loads the front panel, the code, if it matches the platform, and the data for the VI into memory. If the VI needs to be recompiled because of a change in platform or in the interface to a subVI, LabVIEW also loads the block diagram into memory.

LabVIEW also loads the code and data space of subVIs into memory. Under certain circumstances, LabVIEW also loads the front panel of some subVIs into memory. For example, if the subVI uses Property Nodes, LabVIEW must load the front panel because Property Nodes manipulate the characteristics of front panel controls.

As shown in the following example, you can save memory by converting some of your VI components into subVIs. If you create a single, large VI with no subVIs, the front panel, code, and data for that top-level VI end up in memory. If the VI is broken into subVIs, the code for the top-level VI is smaller, and only the code and data of the subVIs is in memory. In some cases, you might actually see lower run-time memory use.

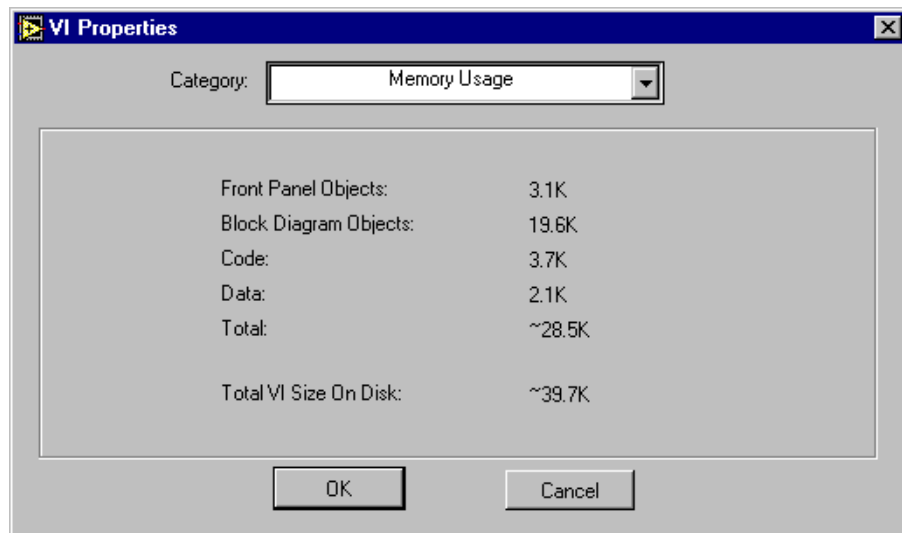


E. Optimizing VI Memory Use

This section describes specific issues about using VI memory more efficiently. We will discuss how to monitor and improve your memory use and suggest ways to efficiently assemble and process arrays and data structures.

How to Monitor Memory Use

As shown in the following example, select **Memory Usage** from the top pull-down menu in the **File»VI Properties** dialog box to get a breakdown of the memory usage for a specific VI. The left column summarizes disk use and the right column summarizes how much RAM is used for the various components of the VI. Notice that the information does not include memory use of subVIs.



You also can use the **Profile** window to monitor the memory used by all VIs in memory.



Note When monitoring VI memory use, be sure to save the VI before examining its memory requirements. The LabVIEW Undo feature makes temporary copies of objects and data, which can increase the reported memory requirements of a VI. Saving the VI purges the copies generated by Undo, resulting in accurate reports of memory information.

General Rules for Better Memory Use

Use the following rules to create VIs that use memory efficiently.

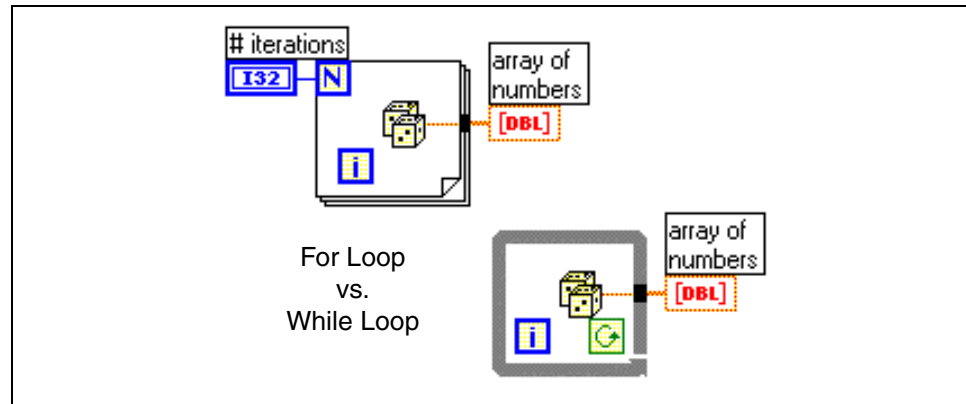
- Breaking a VI into subVIs usually improves memory use because LabVIEW can reclaim subVI data memory when the subVI is not executing.
- Do *not* overuse global and local variables to store arrays or strings; reading a global or local variable generates a copy of the data stored in the variable.
- On open front panels, display large arrays and strings only when necessary. Indicators on open front panels retain a copy of the data that they display.
- If the front panel of a subVI will not be displayed, do not leave unused Property Nodes on the subVI. Property Nodes cause the front panel of a subVI to remain in memory, which increases memory use.
- Do not use the Suspend Data Range option on time- or memory-critical VIs. The front panel for the subVI needs to be loaded for range checking, and extra copies of data are made for the subVI controls and indicators.
- When designing block diagrams, watch for places where the size of an input is different from the size of an output. For example, if you frequently increase the size of an array or string using the Build Array or Concatenate Strings function, you generate copies of data, increasing the number of memory allocations LabVIEW must perform. These operations can fragment memory.
- Use consistent data types for arrays and watch for coercion dots when passing data to subVIs and functions. When LabVIEW changes data types, the output is a new buffer.
- Do *not* use complicated, hierarchical data types, for example, arrays of clusters containing large arrays or strings, or clusters containing large arrays or strings. The [Simple vs. Complicated Data Structures](#) section later in this lesson contains more information about designing efficient data types.

Assembling and Processing Arrays

When designing the block diagram, there are several steps you can take to make your VI use memory more efficiently. For example, you can assemble and process arrays in ways that minimize the amount of memory access required.

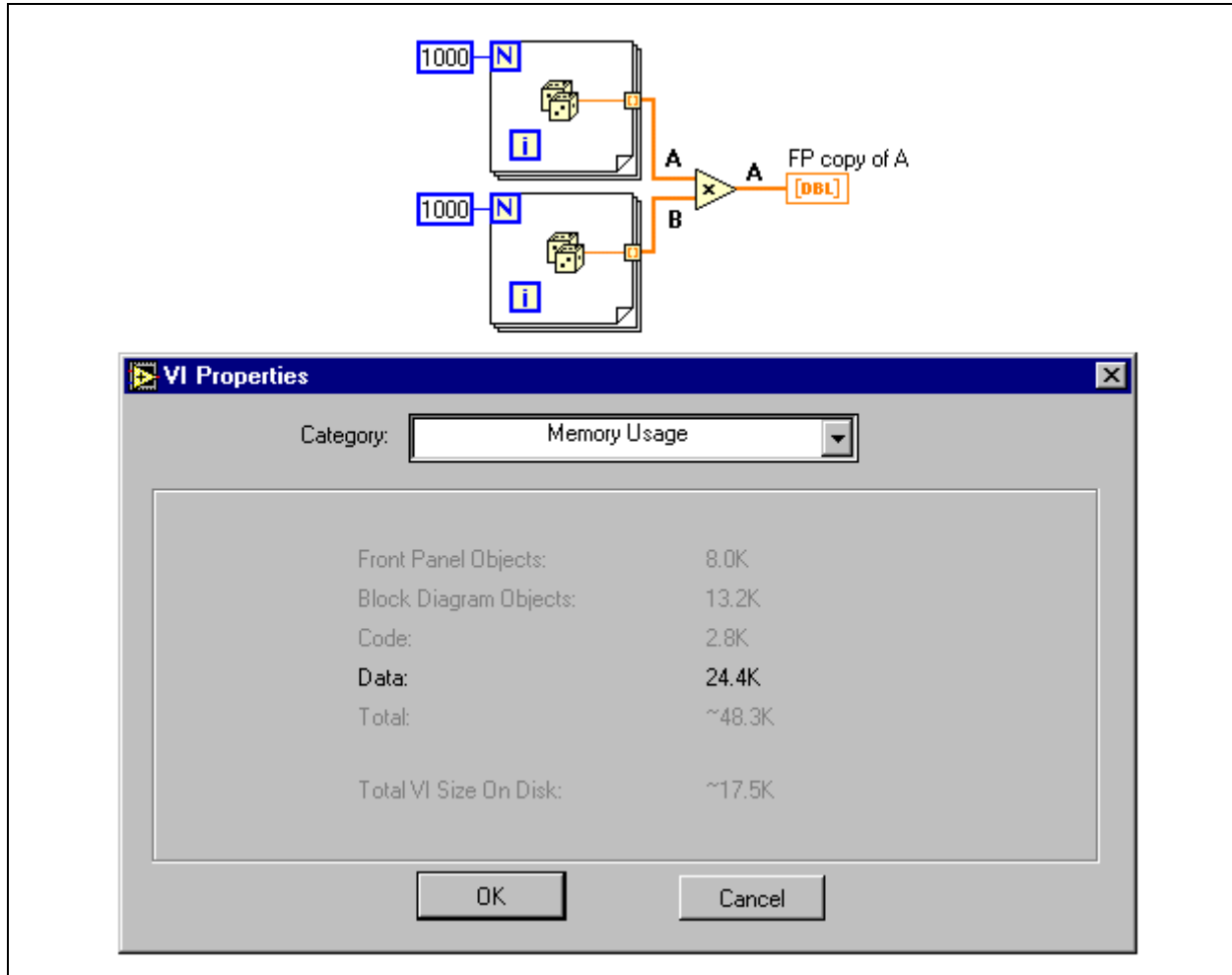
LabVIEW stores arrays of numeric elements in contiguous blocks of memory. If you use For Loops to assemble these arrays, LabVIEW can determine the amount of memory needed and allocate the necessary space prior to the first iteration. However, if you use While Loops, LabVIEW

cannot predetermine the space requirements you will need. LabVIEW might need to relocate the entire buffer as the array grows in size, perhaps several times. The time needed for relocation increases with the size of the array. Therefore, you should use For Loops to assemble arrays when possible, rather than using While Loops or concatenating arrays with the Build Array function.

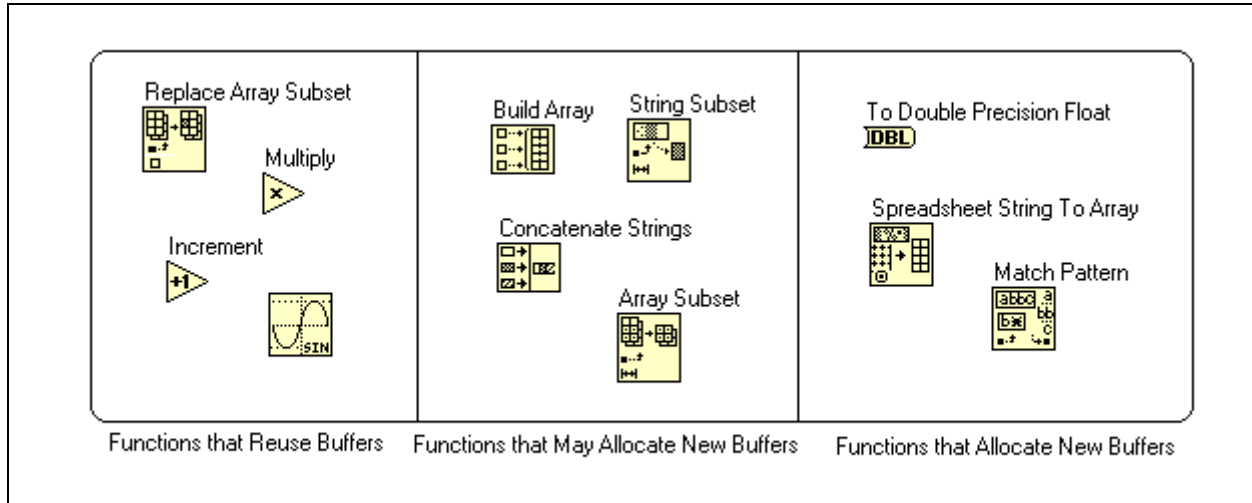


Inplaceness

When possible, LabVIEW's compiler reuses a function's input buffers to store its output. This buffer sharing is called *inplaceness*. In the following example, the Multiply function output uses the same buffer as the top input. The array at the right is said to be in place to array A.



A function output reuses an input buffer if the output and the input have the same data type, representation, and, in arrays, strings, and clusters, the same structure and number of elements. Functions capable of inplaceness include Trigonometric and Logarithmic functions, most Numeric functions, and some string and array functions such as To Upper Case and Replace Array Element. A function that shifts or swaps elements of an array, such as Replace Array Element, can reuse buffers. Some functions such as Array Subset and String Subset might not copy data but might pass pointers to the subarrays or substrings. In the following illustration, A, B, C, and D identify individual buffers. Build Array and Concatenate Strings are special functions. They operate inplace when they can, but sometimes they must allocate new buffers.



Coercion and Consistent Data Types

Consider the Random Number (0-1) function, commonly used in the examples shown in this course. This function produces double-precision, floating-point (DBL) data. Therefore, DBL arrays are created at the border of For Loops. To save memory, you might consider using single-precision floating-point (SGL) arrays instead of DBL arrays. Of the following three methods to create these SGL arrays, one is correct and two are incorrect. Recall that each DBL value requires 8 bytes of memory, while each SGL value requires 4 bytes of memory.

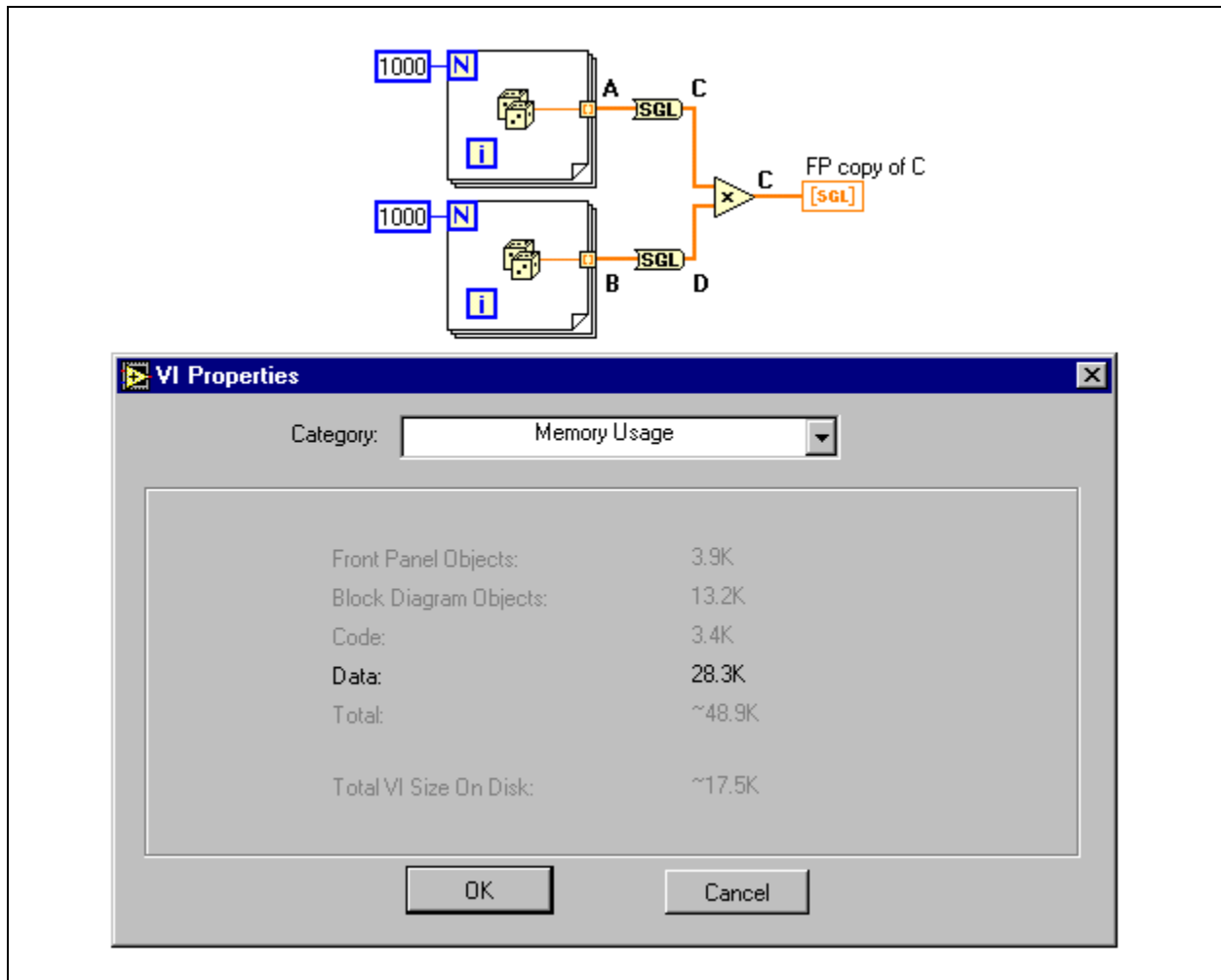
Method 1 (Incorrect)

The diagram illustrates a LabVIEW VI (Virtual Instrument) and its memory usage. The VI consists of two sub-VI icons, each with a '1000' value and an 'N' terminal. These are connected to a multiplication function block (represented by a triangle with an 'x'). The outputs of the multiplication function are labeled 'A' and 'B'. The output 'A' is connected to a front panel control labeled 'FP copy of A' with an '[SGL]' terminal. The output 'B' is connected to a terminal labeled 'C'. Below the diagram is the 'VI Properties' dialog box, which shows the following memory usage statistics:

Category:	Memory Usage
Front Panel Objects:	3.9K
Block Diagram Objects:	13.2K
Code:	3.1K
Data:	24.3K
Total:	~44.5K
Total VI Size On Disk:	~17.5K

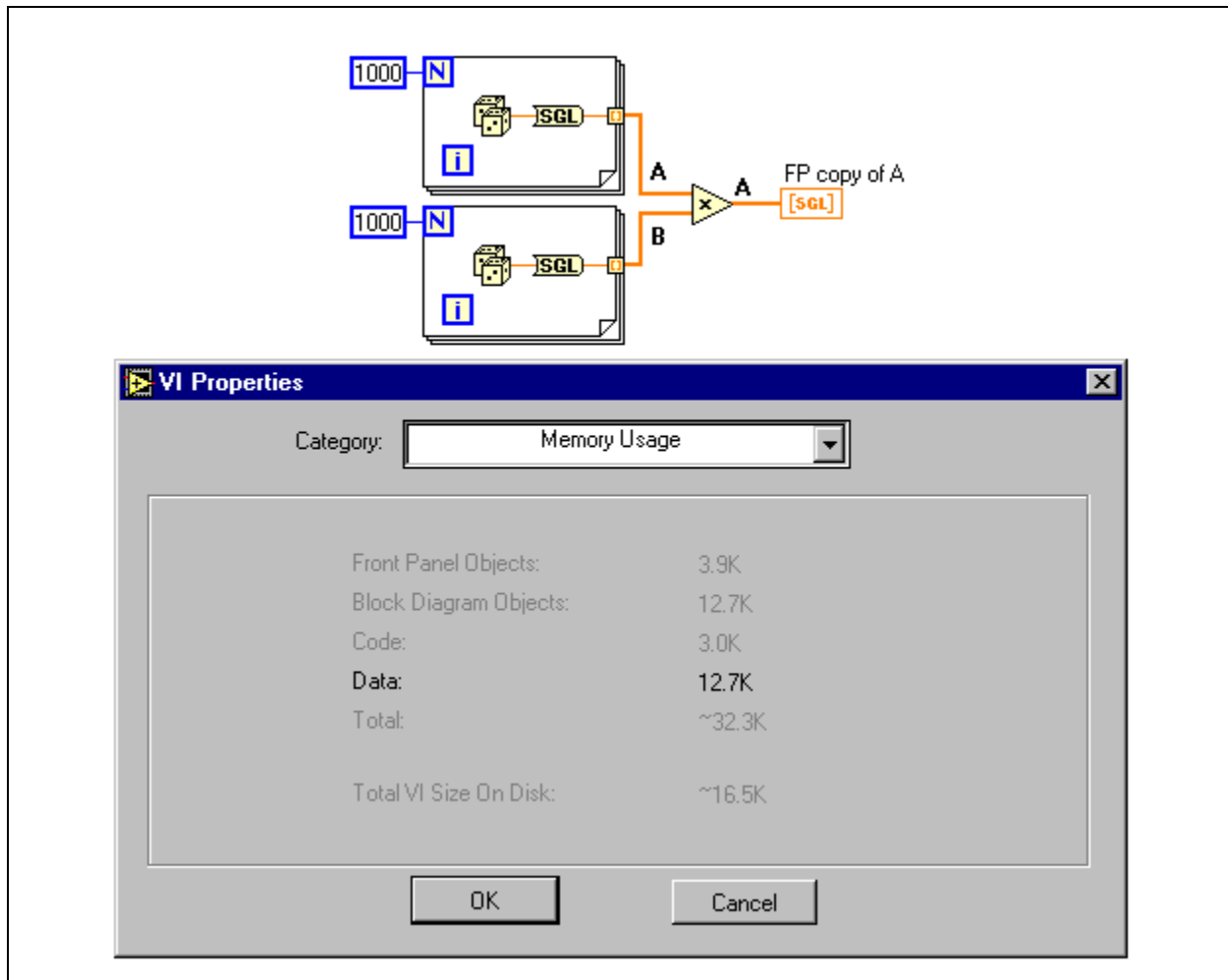
Method 1 might be your first attempt to save data space memory. You might think changing the representation of the array on the front panel (FP) to SGL can save space memory. However, this method does not affect the amount of memory needed by the VI, because the function creates a separate buffer to hold the converted data. The coercion dot on the SGL array terminal indicates the function created a separate buffer to hold the converted data.

Method 2 (Incorrect)



Method 2 is an attempt to remove the coercion dot by converting each array to SGL using the To Single Precision Float function located on the **Functions»Numeric»Conversion** palette. However, this method also increases the size of the data space, because the function creates two new buffers, C and D, to hold the new SGL arrays.

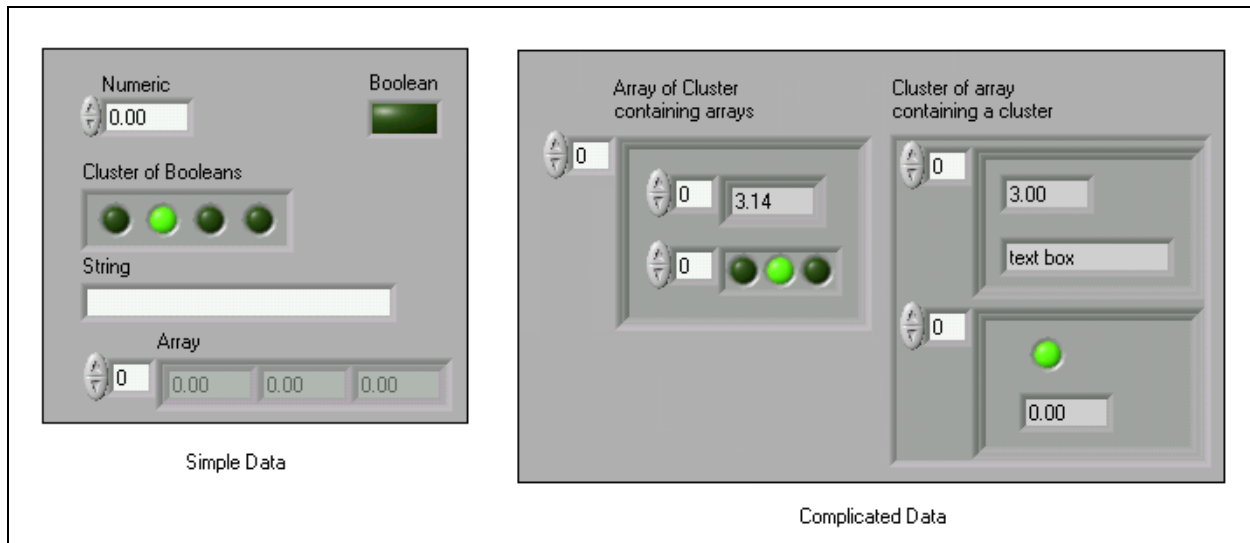
Method 3 (Correct)



Method 3 reduces the size of the data space considerably, from 24.8 KB to 12.8 KB. This method converts the Random Number (0-1) function output to SGL *before* the array is created; therefore, this method creates two SGL arrays at the border of a For Loop rather than two DBL arrays.

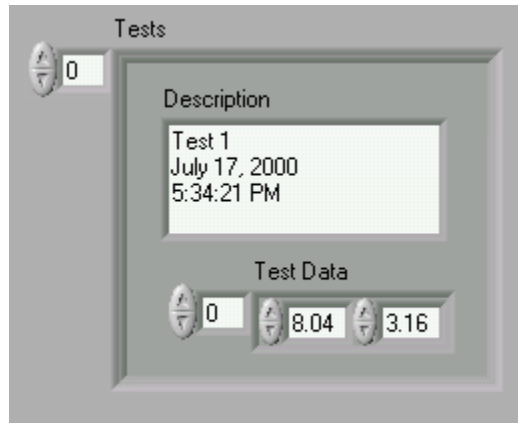
Simple vs. Complicated Data Structures

Simple data types, which include strings, numbers, Boolean data types, clusters of numbers or Boolean data types, and arrays of numbers or Boolean data types, are referenced in memory. Other data, referred to as nested, or complicated data, is more difficult to reference. Examples of nested data include arrays of strings, clusters containing arrays of clusters, and arrays of clusters containing arrays.

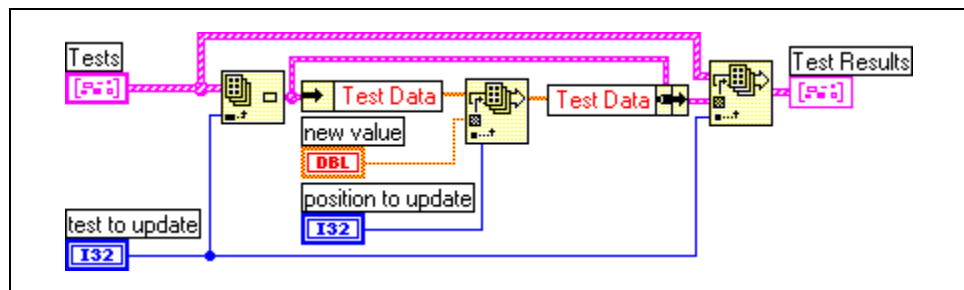


For the best performance, avoid creating complicated data structures. Performance can suffer because it is difficult to access and manipulate the interior elements without generating copies of data. Therefore, keep your data structures as flat as possible. Flat data structures can generally be manipulated easily and efficiently.

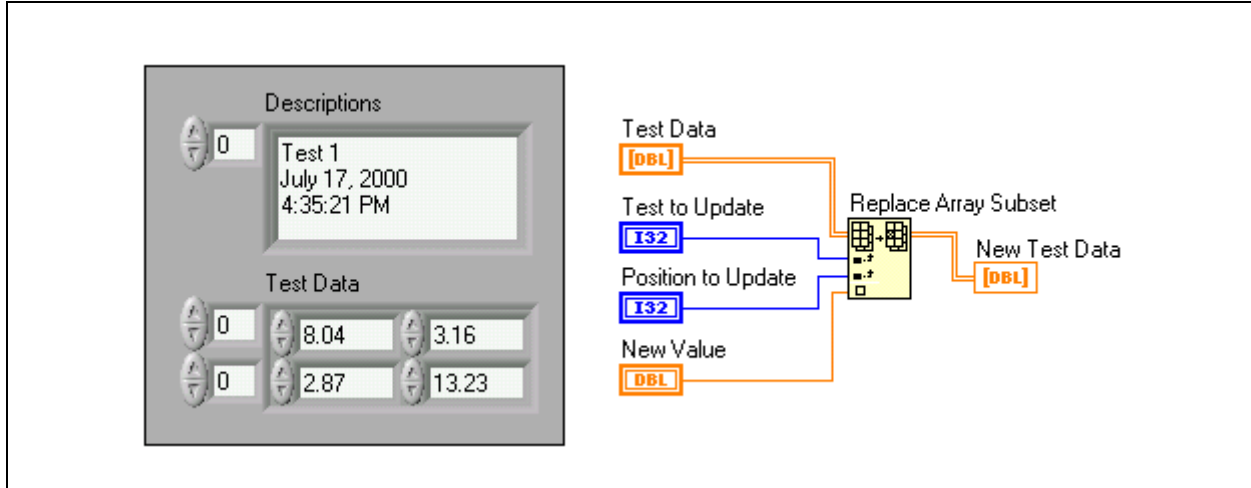
Consider an application in which you want to record the results of several tests. In the results, you want a string describing the test and an array of test results. One data type that you might use to store this information is an array of clusters containing a description string and an array of test data, as shown in the following front panel.



Now, consider what you need to do to change an element in the Test Data array. First, you must index an element of the overall Tests array. For that element, which is a cluster, you must unbundle the elements to get to the array. You then replace an element of the array and store the resulting array in the cluster. Finally, you store the resulting cluster in the original array. An example of this is shown in the following illustration.



Copying data is costly both in terms of memory and performance. The solution is to make the data structures as flat as possible. In this case, you could store the data in two arrays, as shown in the following block diagram. The first array is an array of strings. The second array is a 2D array, where each row is a given test's results. In the following example, the front panel contains new controls to store the data and the block diagram performs the same change to the test data, as shown in the previous block diagram.

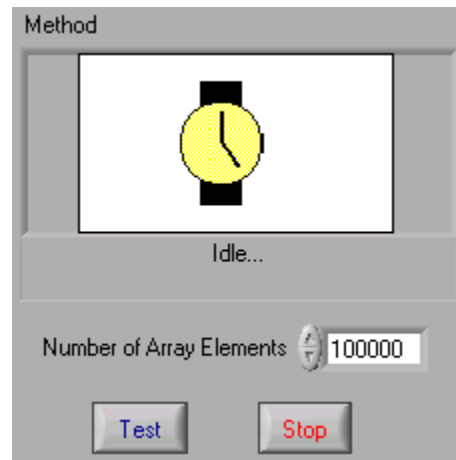


Exercise 6-5 Assembling Arrays VI

Objective: To observe the speed of building arrays using several different methods.

Load and run a VI that creates an array of numbers using several methods. Using the **Profile** window, compare the performance of VIs using For Loops, While Loops, Auto-Indexing, and the Build Array function to create arrays. Also, compare these methods with a technique that uses the Replace Array Element function on an existing array.

Front Panel

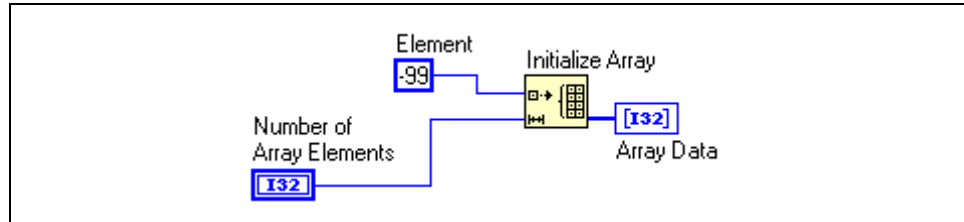


1. Close all other VIs that you might have open.
2. Open the Assembling Arrays VI.
3. To display the **Profile** window, select **Tools»Advanced»Profile VIs**. Select the option to display timing statistics. Click the **Start** button in the **Profile** window to begin a profiling session. Do not gather any memory statistics.
4. Run the Assembling Arrays VI. To test the different array-building methods, click the **Test** button. The Method indicator shows which method is being tested. The VI tests six different methods.
5. After running the test several times, click the **Snapshot** button in the **Profile** window.

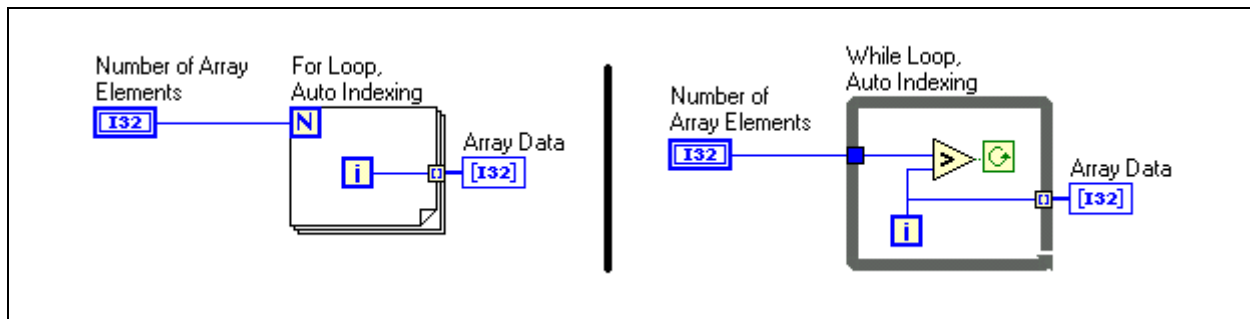
We will now describe the six different array building techniques and why the performance of these VIs differs.

Using the **Profile** window, place a checkmark in the **Timing Statistics** checkbox and notice the average execution times shown.

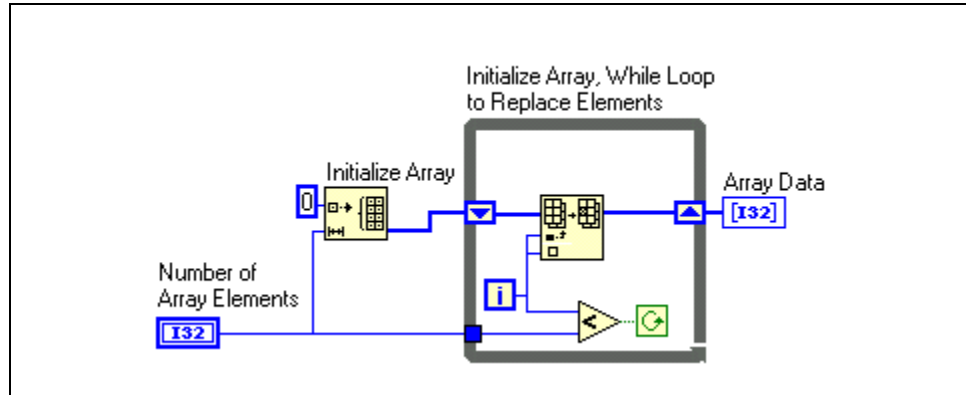
The Initialize Array function is slightly faster than an Auto Indexing For Loop. In both instances, the array size is known and LabVIEW can allocate the correct amount of memory only once. Recall that resizing requires more execution time. Therefore the Initialize Array function or an Auto Indexing For Loop are the best ways to create arrays.



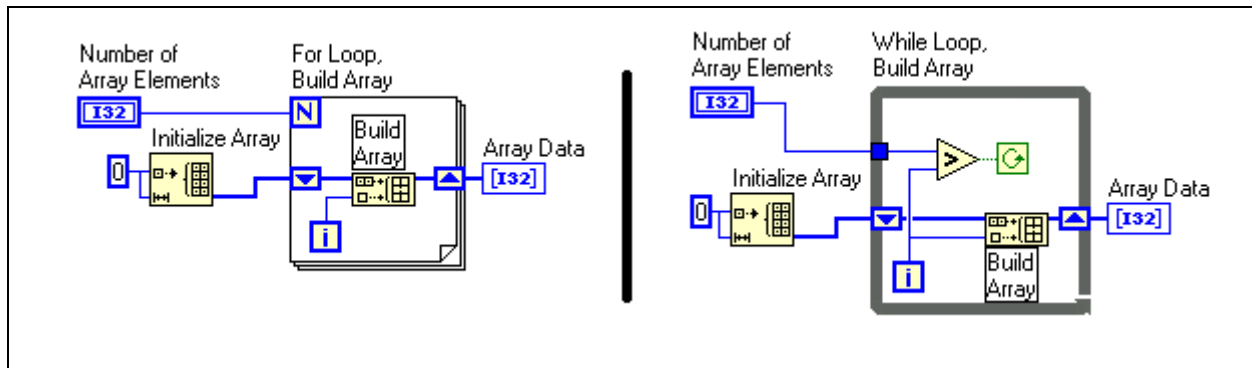
A For Loop executes a predetermined number of times. A While Loop does not, so LabVIEW cannot predetermine how much memory to allocate if an array is built inside a While Loop. When the array is built in a While Loop, it is resized as needed. Each resizing requires the operating system to allocate a new buffer and then copy the contents of the old buffer into the new, larger buffer. The more resizing that occurs, the more time execution takes. A For Loop requires less resizing, so it executes faster than a While Loop. Notice this difference when comparing the Average execution time for the For Loop using the Auto Indexing VI and While Loop using the Auto Indexing VI.



If you can predetermine the maximum size of the array to be built, then you can initialize this array and pass it into the While Loop. Once inside the While Loop, each element can be updated or replaced as needed. This is somewhat efficient, because the array does not need to be reallocated once it is passed into the While Loop. This is still not as efficient as the For Loop or Initialize Array function.



Avoid using the Build Array function inside a loop. Every time a new value is appended to the array, LabVIEW must reallocate the buffer and copy the entire array to the new location. Thus execution times for the Build Array function are the slowest.



6. Close the **Profile** window. Close the VI and do not save changes.

End of Exercise 6-5

Exercise 6-6 Performance Enhancements VI

Objective: To improve the performance of a VI.

The Performance Enhancements VI was created to perform the following tasks.

1. Acquire 1000 points of a waveform from a channel.
2. Save this data to disk.
3. Display the power spectrum of the acquired waveform.

The VI is poorly written. The VI consumes a lot of memory, runs slowly, and is poorly documented. In addition, the VI does not give the desired results for waveforms above 60 Hertz.

This project requires you to utilize information from the previous chapter as well as draw upon other information in the course.

1. Close all VIs you might have open.
2. Open the Performance Enhancements VI in the `Exercises\LV Basics2` directory. Show the **Profile** window by selecting **Tools»Advanced»Profile VIs**. Select **Timing Statistics** in the **Profile** window.
3. Run the VI. After it finishes, click the **Snapshot** button in the **Profile** window to view execution information about the VI. Notice how many times VIs are called and which VIs being called multiple times are likely to be slow. Notice which VIs were run many times and observe which VIs you guess are likely to take longer execution times.
4. Select **File»VI Properties** and select **Memory Usage** from the top pull-down menu. A well written VI for this application should consume a total of about 25k of memory. Notice how much this poorly written VI consumes.
5. Modify the VI to so that this application:
 - Runs correctly for higher frequency waveforms.
 - Runs faster.
 - Uses better data structures and is easier to understand.
 - Is better documented.

Use the tips in this lesson, information from other chapters in this course, and the following tips to get started:

- Putting High Level VIs in loops creates redundant function calls.
- Gathering or plotting data one point at a time is less efficient than acquiring multiple points.

- Flat data types are more memory efficient than complex data structures.
 - There are a variety of ways to document a VI.
6. Save the modified VI.
 7. Run the VI. Notice improvements in run time and memory usage. Also notice the improvements in readability and simplicity of code.



Note Make sure you save the VI before you examine its memory usage. This prevents the LabVIEW Undo feature from allocating memory unnecessarily.

8. When you are satisfied with the VI, save it as `Performance Improvements.vi`.

End of Exercise 6-6

Exercise 6-7 Performance Challenge VI/ Maximized Performance VI

Objective: To improve the performance of a VI.

The Performance Challenge VI performs the following tasks.

1. Generates four runs of data. Each run should consist of an array of 5,000 data points and a string containing a time stamp of when the data were taken. Each element needs only digits of precision.
2. Saves all acquired data and associated time stamps to disk. The file will be used only in LabVIEW applications, so it does not need to be in any specific format.
3. Finds the maximum value in each run of data and plot 100 points around the maximum value to a waveform graph. The final graph should contain four traces, one for each run of data.

This VI is poorly written. It runs slowly and takes more memory than necessary. In this exercise, optimize the VI to increase its run speed while decreasing its memory requirements.

1. Close all VIs you might have open.
2. Open the Performance Challenge VI. Show the **Profile** window by selecting **Tools»Advanced»Profile VIs**. Before running Performance Challenge, select **Timing Statistics** and **Timing Details** in the **Profile** window so that you can gain a better understanding of the VI run time.
3. Run the VI. After it finishes, click the **Snapshot** button in the **Profile** window to view execution information for the VI. When looking at this information, notice how much time is taken updating local variables, and also how many times this application calls the file I/O subVIs.
4. Select **File»VI Properties**. Select **Memory Usage** from the top pull-down menu. Notice the considerable amount of memory taken by the data in this VI.
5. Modify the VI to improve the run speed and reduce the amount of memory required. Use the tips in this lesson and the following guidelines to get you started:
 - Local variables increase memory requirements and slow run speed, especially when they are accessed in loops.
 - Create arrays in an efficient manner.
 - Minimize the amount of file I/O operations performed in an application. Only open and close a file when necessary.
 - Avoid unnecessary computations and data conversions, especially in loops.

- Minimize and simplify front panel displays.
- Use simple data structures.



Tip You only need to meet the requirements of the application described previously. How you choose to implement this application is up to you.

6. Save the modified VI.
7. Run the VI, noticing the speed improvements in the **Profile** window and the memory use improvements in **VI Properties**. You might want to repeat steps 5 through 7 several times, further optimizing the application.



Note Make sure you save the VI before you examine its memory requirements. This prevents the LabVIEW Undo feature from allocating memory unnecessarily.

8. When you are satisfied with the VI, save it as `Maximized Performance.vi`.

End of Exercise 6-7

Summary, Tips, and Tricks

Depending on the operating system, LabVIEW uses either multithreading or co-operative multitasking to perform tasks simultaneously. With multithreading, different tasks can use different execution threads, whereas in a co-operative multitasking system, different tasks use the same execution thread.

Use the **Profile** window to gauge VI performance and to help you locate the tasks that require most of your VI's run time and memory use. You can then concentrate on improving those troublesome areas to enhance the overall performance of your VI.

Use the following tips to speed up your VIs:

- Reduce the number of I/O calls you make by reducing the amount of data you acquire, or by acquiring more data in fewer calls.
- Reduce the number of controls and indicators you have on the front panel.
- Avoid using autoscaling on graphs and charts.
- Update graphs and charts with several points at a time, not one point at a time.
- Force less important parallel tasks to wait, using the Wait (ms) function, so more crucial ones have more processor time.
- Avoid unnecessary computation in looping structures.

Although LabVIEW removes much of the difficulty in managing computer memory, you also have less control over that process. Remember, you can monitor your computer's memory by accessing the Memory Usage page in the **File»VI Properties** dialog box and in the **Tools»Advanced»Profile VIs** window.

Virtual memory has advantages and disadvantages. It can significantly increase the amount of RAM available for LabVIEW and your VIs. However, because that RAM is located on the hard drive, performance suffers when it is accessed.

Breaking large top-level VIs into several smaller subVIs reduces the amount of memory consumed in your application and can improve performance.

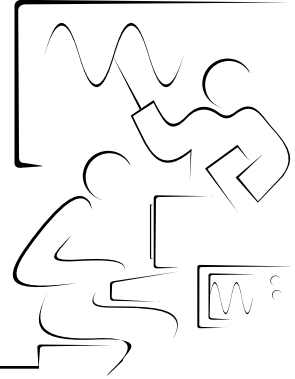
Use the following tips to improve the overall performance of your VIs:

- Avoid overusing local and global variables.
- Avoid displaying and manipulating large arrays and strings.
- Use functions that reuse data buffers.

- Use consistent data types. In other words, avoid coercion dots.
- Use simple data structures that are as flat as possible.
- When generating arrays inside loops whose representation must be changed, change the representation inside the loop, not after.

Notes

Appendix

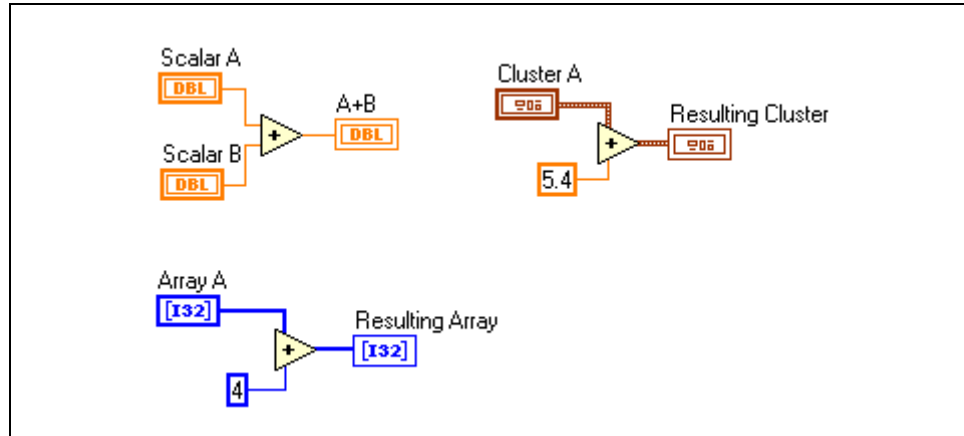


This appendix contains the following sections of useful information for LabVIEW users:

- A. Polymorphic subVIs
- B. Custom Graphics in LabVIEW
- C. The LabVIEW Web Server
- D. Additional Information
- E. ASCII Character Code Equivalents Table

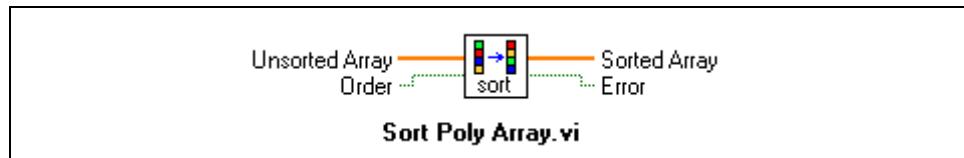
A. Polymorphic SubVIs

LabVIEW built-in functions can handle different types of data for the same terminal, a capability called polymorphism. An example of polymorphism is the Add function.



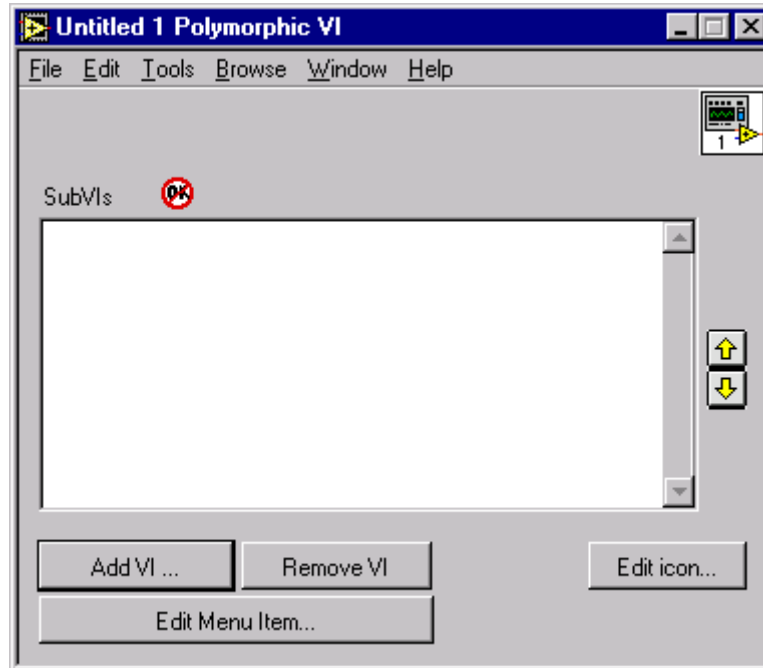
It is also possible to create your own polymorphic VIs, which can handle more than one type of data for the same terminal. These polymorphic VIs can then be used as polymorphic subVIs.

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. 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.



Complete the following steps to create your own polymorphic VI.

1. Create a set of VIs having the same connector pane pattern, one for different sets of data types.
2. Create a polymorphic VI by selecting **File»New** and selecting **Polymorphic VI** in the **New** dialog box. The **Polymorphic VI** builder dialog box appears.



3. Add each of the set of VIs to the polymorphic VI using the **Add VI** button.
4. 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.



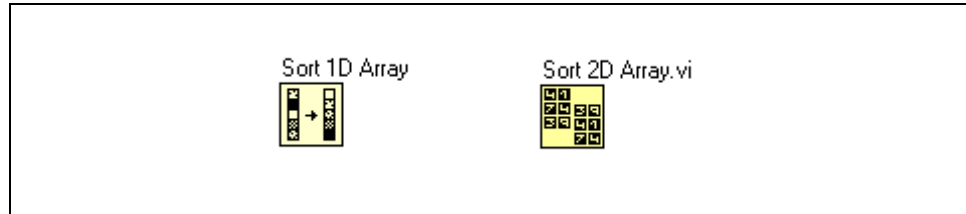
Note Context help for the polymorphic VI is not associated with context help for the VIs that compose the polymorphic VI. Therefore, you must create new context help by using the **Documentation** page in the **File»VI Properties** dialog box.

A polymorphic VI is a collection of subVIs with the same connector pane patterns. Each subVI is an instance of the polymorphic VI. When data are wired to a polymorphic subVI, LabVIEW automatically chooses which instance to use based on the types of data wired to the inputs of the polymorphic subVI. If the instance is not available, a broken wire appears. You can override LabVIEW's automatic selection by right-clicking the polymorphic subVI and selecting a specific instance VI from the **Select Type** pull-right menu.

Exercise A-1 Sort Poly Array

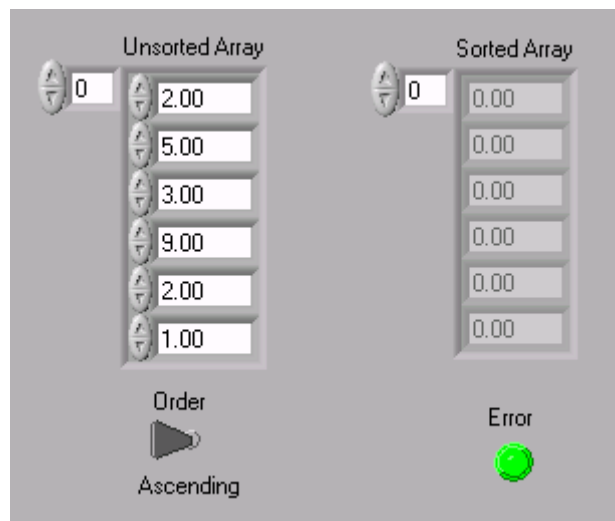
Objective: To create a polymorphic VI and use it as a subVI.

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, 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.

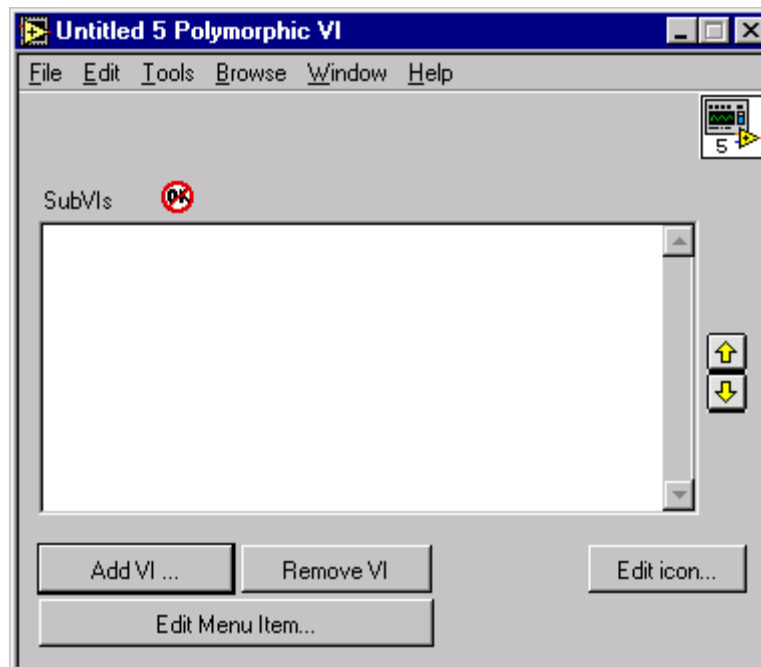
3. Save your work as Sort 1D Array+.vi.
4. The Sort 2D Array function is already built for you. However, you need to complete the icon connector pane. This example was downloaded from ni.com\support in the Example Programs Database. Open this file from exercises\LV Basics2\Sort 2D Array.vi.
5. Create 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.

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 **Polymorphic VI**. If you are at the welcome screen, select **Polymorphic VI** from the **New VI** pull-down menu.



8. Add the Sort 1D Array+ and Sort 2D Array+ VIs 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 your VI as `Sort Poly Array.vi`.
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, *Designing Front Panels*. Through careful use of the decorations, you can increase the readability of the front panels.

Importing Graphics

You can import graphics from other VIs for use as background pictures, items in ring controls, or parts of other controls. However, before you can use a picture in LabVIEW, you need to load it into the LabVIEW Clipboard. There are one or two ways to do this, depending on your platform.

- **Windows**—If you can copy an image directly from a paint program to the Windows Clipboard and then change to LabVIEW, LabVIEW automatically imports the picture to the LabVIEW Clipboard. You also can use the **Import Picture from File** option from the LabVIEW **Edit** menu to import a graphics file into the LabVIEW Clipboard. LabVIEW recognizes graphics files in the following formats: CLP, EMF, GIF, PCX, BMP, TARGA, TIFF, LZW, WFM, and WPG.
- **Macintosh**—If you copy from a paint program to the Clipboard and then change to LabVIEW, LabVIEW automatically imports the picture to the LabVIEW Clipboard.
- **UNIX**—You can use the **Import Picture from File** option from the UNIX **Edit** menu to import a picture of type X Window Dump (XWD), which you can create using the xwd command.

After a picture is on the LabVIEW Clipboard, you can paste it as a static picture on the front panel, or you can use the **Import Picture** option of a shortcut menu, or the Import Picture options in the Control Editor.

Custom Controls

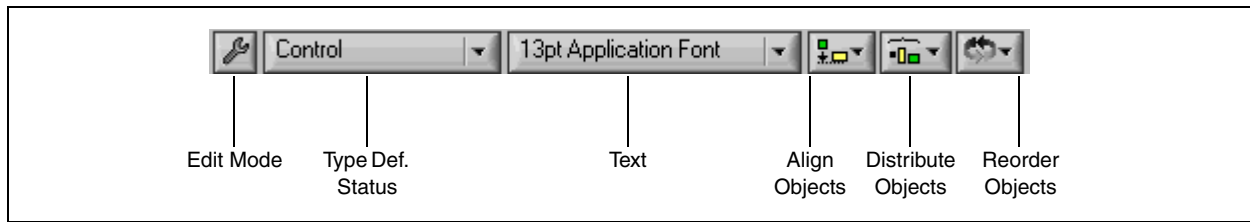
You can customize LabVIEW controls and indicators to change their appearance on the front panel. You also can save these controls for use in other VIs. Programmatically, they function the same as standard LabVIEW controls.

Control Editor

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.

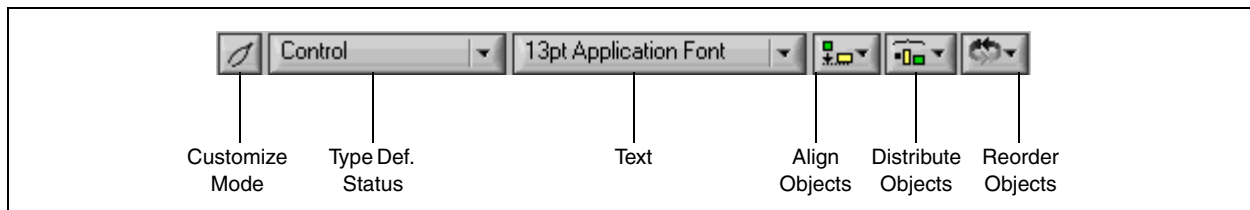
Edit Mode

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



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**.



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 showing 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.

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 everywhere you use it. In addition to data type, its size, color, and appearance must also 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.

Saving Controls

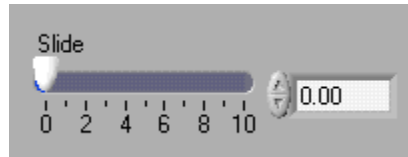
After creating a custom control, you can save it for use later. By default, controls saved on disk has a `.ctl` extension. You also can place controls in the **Controls** palette using the same method as that you used to add subVIs to the **Functions** palette.

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.

Exercise A-2

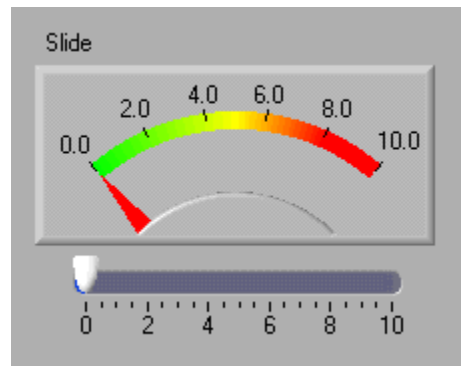
Objective: To use the Control Editor to modify a control.

1. Open a new front panel.
2. Place a Horizontal Pointer Slide located on the **Controls»Numeric** palette on the front panel. Right-click the slide and select **Visible Items»Digital Display**.

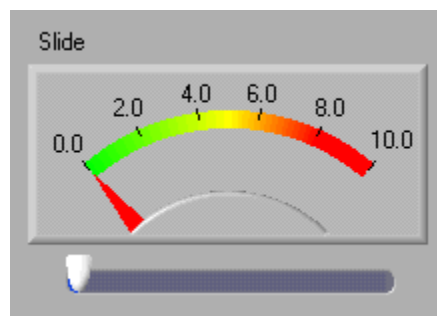


Modifying the Control

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`, 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.

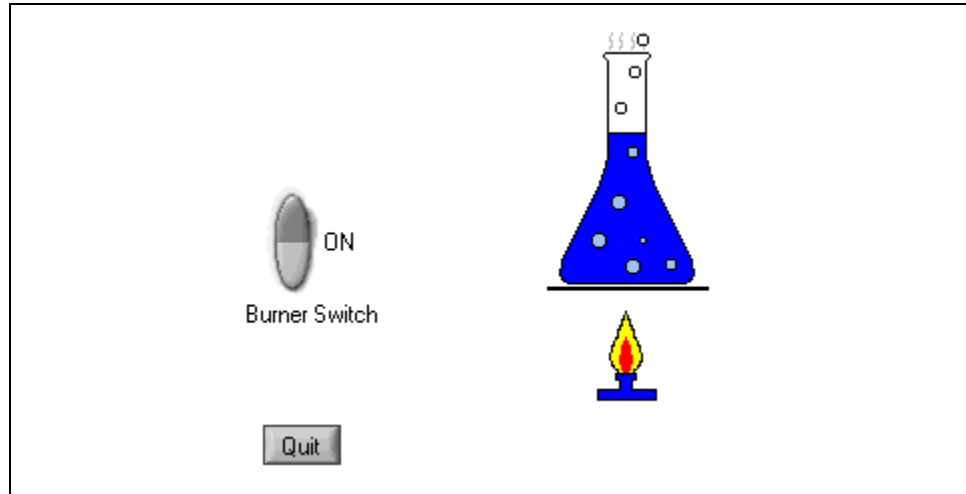
End of Exercise A-2

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 in `exercises\Basics2` 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»Classic Controls»Boolean** 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.

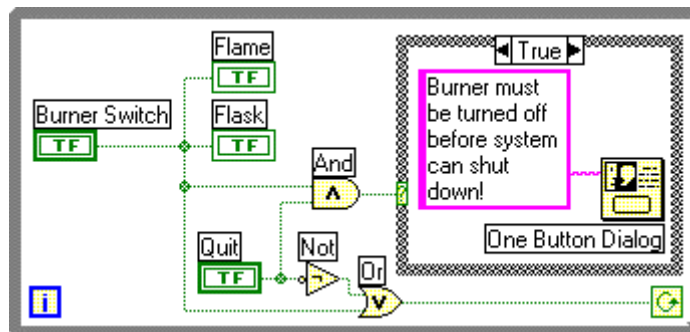


Note The default state of the LED is FALSE. If you do not see the picture, the LED is probably in the FALSE state.

- 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>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. The LabVIEW Web Server

The Web is one of the most popular ways to share information, and it is a good way to share your measurement and automation data. LabVIEW contains several methods to help you publish your data to the web and create HTML documents. Lesson 3 describes DataSocket and how it uses the URL method to describe and access LabVIEW data.

Another method is to use the LabVIEW Web Server to publish images of your VI front panels on the Web. By default, after you enable the Web Server all VIs are visible to all Web browsers. However, you can control browser access to the published front panels and configure which VIs are visible on the Web. To display VI front panels on the Web, the VIs must be in memory on your computer.

To configure the Web Server, select **Tools»Options** and select **Web Server: Configuration** from the top pull-down menu.

The **Web Server: Configuration** window is where you enable the Web Server. The default Web Server configuration is suitable for most applications. If you need to change this default configuration, refer to the LabVIEW help.

The **Web Server: Browser Access** window allows you to configure which browser addresses can view your VI front panels. 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. Refer to the Context Help window for more information about the syntax used to specify browser access.

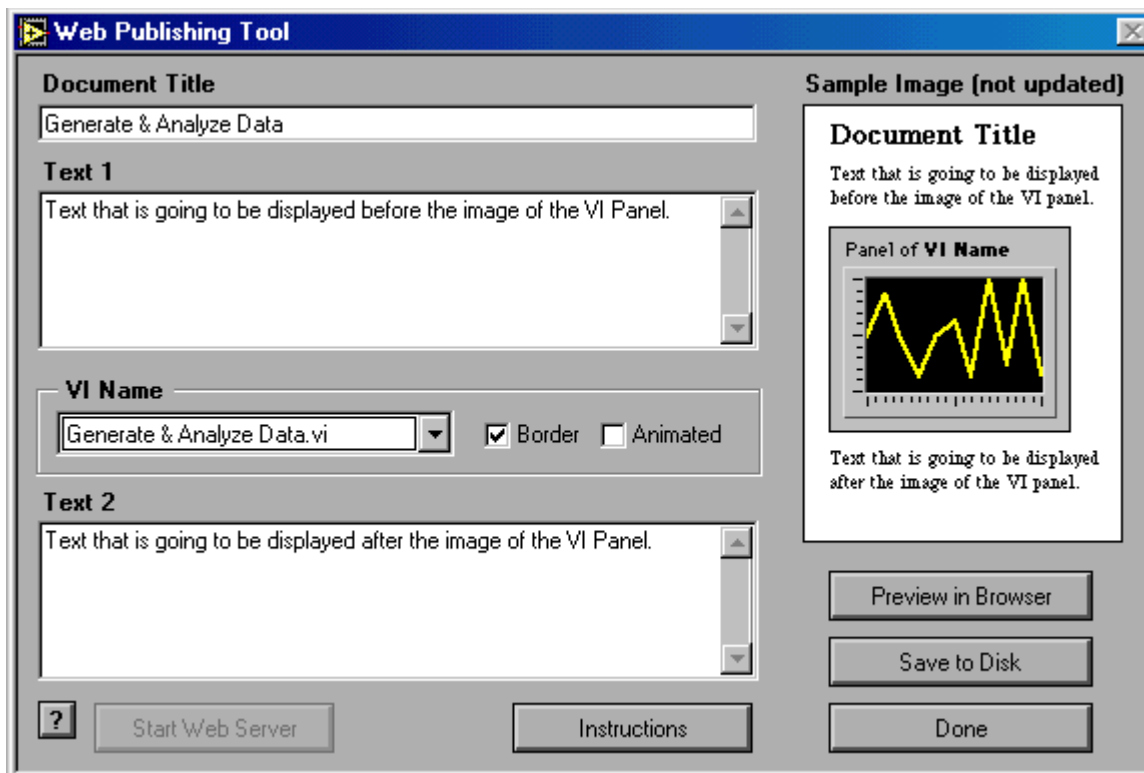
The **WebServer: Visible VIs** window lets you specify which VIs are accessible to a Web browser. When a Web browser attempts to obtain a VI front panel image, the Web Server compares the VI name to the entries in the Visible VIs list to determine whether it should grant access. If an entry in the Visible 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. Refer to the Context Help window for more information about the syntax used to specify visible VIs.



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 for more information about this toolset.

The LabVIEW Web Publishing Tool

You also can use the **Tools»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 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.



Note To display VI front panels on the Web, the VIs must be in memory on your computer.

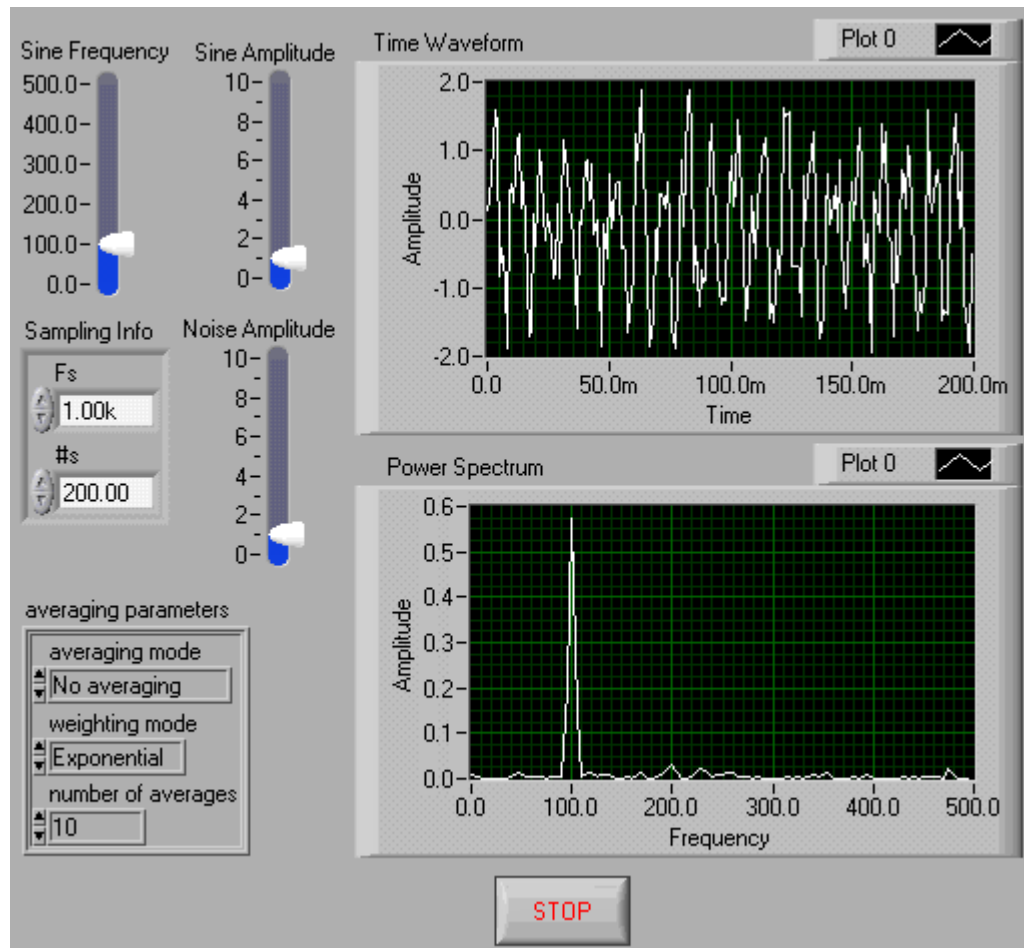
You can preview the document in your default browser by clicking the **Start Web Server** button and then clicking the **Preview in Browser** button. If the **Start Web Server** button is dimmed, the Web Server is already running. When you click the **Save to Disk** button, the title, text, and VI front panel image are saved 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`. The **Document URL** dialog box appears with the URL for your HTML document.

Exercise A-4 Generate & Analyze Data

Objective: To use the LabVIEW Web Server tools to display a front panel in a web browser.

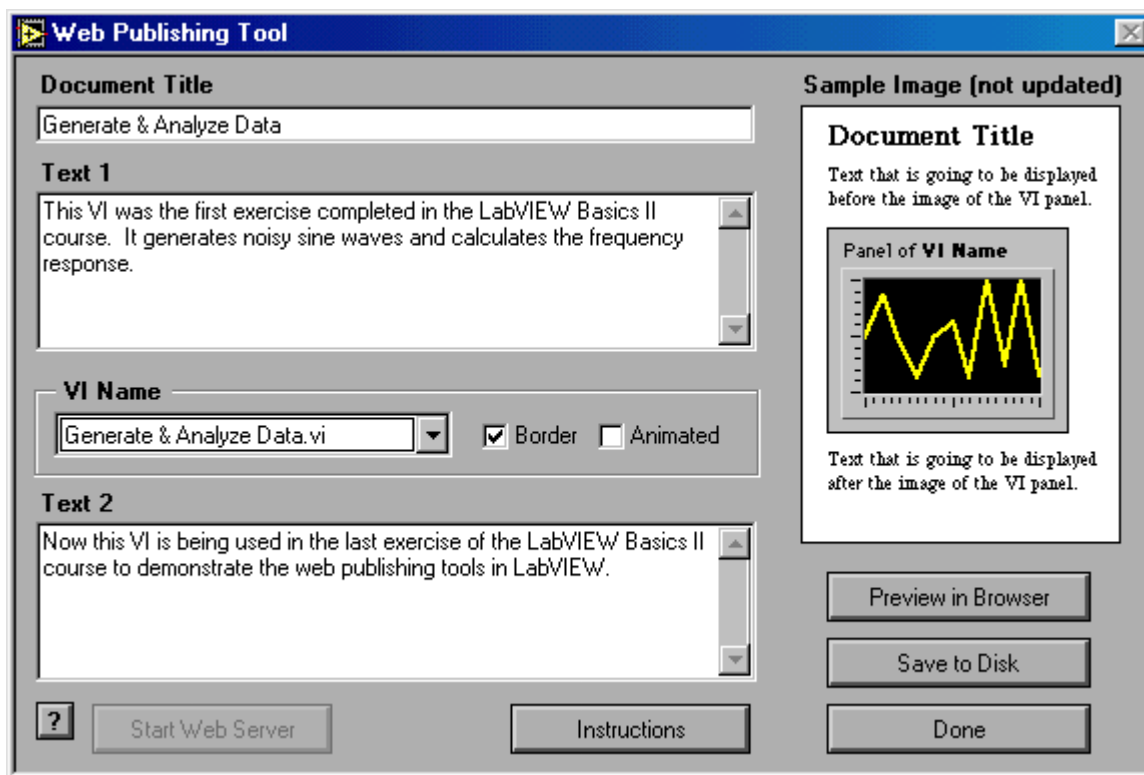
Open the VI you built in Exercise 1-1, the *Generate & Analyze Data VI*, and save its front panel into an HTML document.

Front Panel

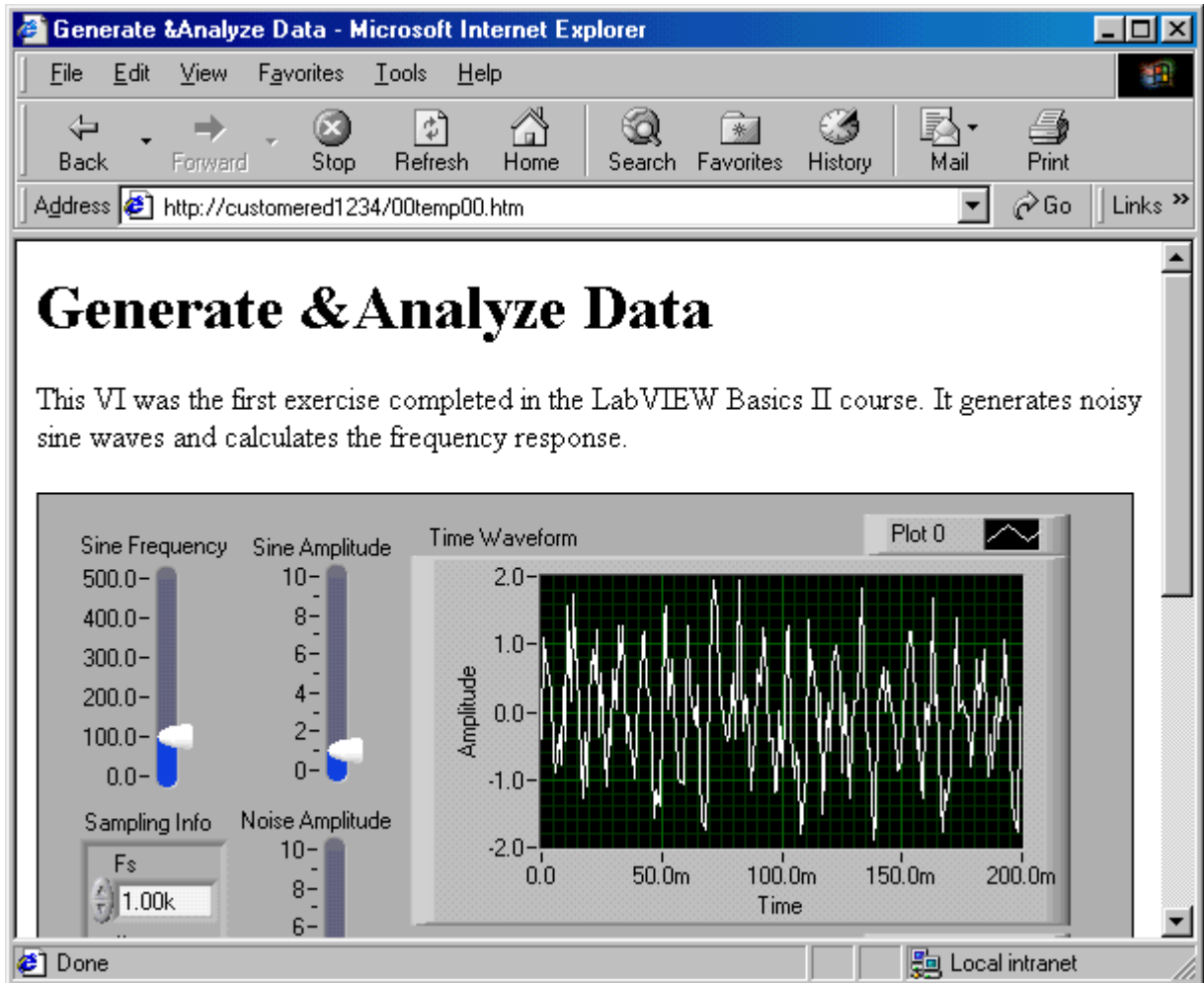


1. Open the Generate & Analyze Data VI from the exercises\LV Basics2 directory.
2. Enable and configure the Web Server by selecting **Tools»Options**.
3. Select **Web Server: Configuration** from the pull-down menu at the top of the **Options** dialog box and place a checkmark in the **Enable Web Server** checkbox.
4. Click the **OK** button to close the **Options** dialog box. The LabVIEW Web Server is now running.

5. Run the VI for a few seconds and stop it.
6. Select **Tools»Web Publishing Tool**. Click the **Instructions** button and read how to use this tool.
7. Click the **continue** link and configure the following window.

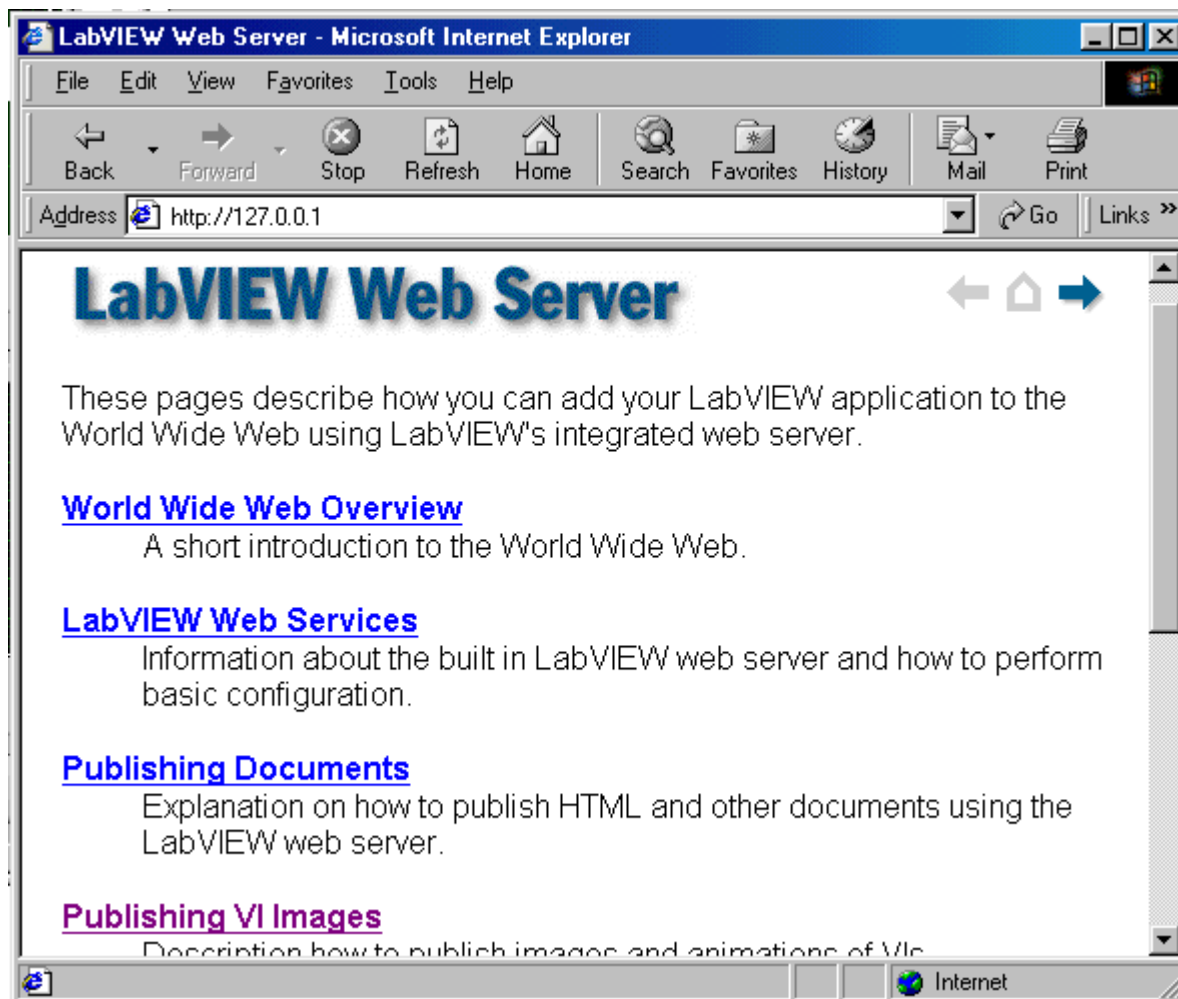


8. Click the **Preview in Browser** button to open and display the front panel in a web browser. A window similar to the one shown in the next figure appears.



Note The Animated option only works with Netscape Navigator. You can only view static images using Internet Explorer.

9. 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 `Generate & Analyze Data.htm` in the LV Basics 2 directory. A warning message appears, telling you where to install the file if you want other machines to view it.
10. Click the **Done** button to exit the Web Publishing Tool.
11. Go back to the Browser application and type the following URL into the address: `http://127.0.0.1`. This specifies the local machine. The following window appears.



12. Read the information and close the browser when you are finished.
13. Turn off the Web Server by selecting **Tools»Options** to display the **Options** dialog box.
14. Select **Web Server: Configuration** from the top pull-down menu of the **Options** dialog box and remove the checkmark from the **Enable Web Server** checkbox.
15. Click the **OK** button to close the **Options** dialog box.
16. Close the Generate & Analyze Data VI.

End of Exercise A-4

D. 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 way to get 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/custed. These courses will 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 might also 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.

E. ASCII Character Code Equivalents Table

The following table contains the hexadecimal, octal, and decimal code equivalents for ASCII character codes.

Hex	Octal	Decimal	ASCII
00	000	0	NUL
01	001	1	SOH
02	002	2	STX
03	003	3	ETX
04	004	4	EOT
05	005	5	ENQ
06	006	6	ACK
07	007	7	BEL
08	010	8	BS
09	011	9	HT
0A	012	10	LF
0B	013	11	VT
0C	014	12	FF
0D	015	13	CR
0E	016	14	SO
0F	017	15	SI
10	020	16	DLE
11	021	17	DC1
12	022	18	DC2
13	023	19	DC3
14	024	20	DC4
15	025	21	NAK
16	026	22	SYN
17	027	23	ETB

Hex	Octal	Decimal	ASCII
20	040	32	SP
21	041	33	!
22	042	34	"
23	043	35	#
24	044	36	\$
25	045	37	%
26	046	38	&
27	047	39	'
28	050	40	(
29	051	41)
2A	052	42	*
2B	053	43	+
2C	054	44	,
2D	055	45	-
2E	056	46	.
2F	057	47	/
30	060	48	0
31	061	49	1
32	062	50	2
33	063	51	3
34	064	52	4
35	065	53	5
36	066	54	6
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

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 Course Manual*

Edition Date: September 2000

Part Number: 320629G-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 _____

E-mail 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? National Instruments web site National Instruments Sales Representative
 Mailing Co-worker Other _____

Customer Education Student Profile

Name _____ Title _____
Company _____ Mail Stop _____
Mailing Address _____
City _____ State/Province _____ Country _____ Zip _____
Telephone _____ Fax _____
E-Mail _____
Date _____ Event Location _____

Industry and Application Information

Which industry does your company primarily serve? (check only one)

- | | | | |
|--|---|---|--|
| <input type="checkbox"/> Automotive | <input type="checkbox"/> Industrial systems -
factory floor/integrator | <input type="checkbox"/> Pharmaceutical | <input type="checkbox"/> Test, measurement,
and instrumentation |
| <input type="checkbox"/> Computer | <input type="checkbox"/> Medical | <input type="checkbox"/> Aero/avionics | <input type="checkbox"/> Telecommunications |
| <input type="checkbox"/> Consumer products | <input type="checkbox"/> Military/space | <input type="checkbox"/> Semiconductor | <input type="checkbox"/> University/education |
| <input type="checkbox"/> Electronics | <input type="checkbox"/> Paper/pulp | <input type="checkbox"/> ATE/automated test | |
| <input type="checkbox"/> Graphics | <input type="checkbox"/> Petrochemical/plastics | <input type="checkbox"/> Other _____ | |

If you are currently a customer of National Instruments, please check the products you use:

- | | | | |
|--|--|--------------------------------|---|
| <input type="checkbox"/> LabVIEW™ | <input type="checkbox"/> HiQ™ | <input type="checkbox"/> DAQ | <input type="checkbox"/> Fieldbus™ |
| <input type="checkbox"/> LabWindows/CVI™ | <input type="checkbox"/> ComponentWorks™ | <input type="checkbox"/> SCXI™ | <input type="checkbox"/> IMAQ™ Vision |
| <input type="checkbox"/> BridgeVIEW™ | <input type="checkbox"/> VirtualBench™ | <input type="checkbox"/> GPIB | <input type="checkbox"/> Serial |
| <input type="checkbox"/> Lookout™ | <input type="checkbox"/> Measure™ | <input type="checkbox"/> VXI | <input type="checkbox"/> Motion control |

Please check the operating system(s) you use:

- | | | | |
|-------------------------------------|--------------------------------------|--|--------------------------------|
| <input type="checkbox"/> Windows NT | <input type="checkbox"/> Windows 3.1 | <input type="checkbox"/> Mac OS | <input type="checkbox"/> HP-UX |
| <input type="checkbox"/> Windows 95 | <input type="checkbox"/> Sun | <input type="checkbox"/> Concurrent PowerMax | |

Please check the bus architecture(s) you use:

- | | | | |
|-----------------------------------|------------------------------|------------------------------------|------------------------------|
| <input type="checkbox"/> PC/XT/AT | <input type="checkbox"/> PCI | <input type="checkbox"/> Macintosh | <input type="checkbox"/> DEC |
| <input type="checkbox"/> PCMCIA | <input type="checkbox"/> VME | | |

What other products are of interest to you:

- | | | | |
|---|---|-------------------------------|---|
| <input type="checkbox"/> LabVIEW | <input type="checkbox"/> HiQ | <input type="checkbox"/> DAQ | <input type="checkbox"/> Fieldbus |
| <input type="checkbox"/> LabWindows/CVI | <input type="checkbox"/> ComponentWorks | <input type="checkbox"/> SCXI | <input type="checkbox"/> IMAQ Vision |
| <input type="checkbox"/> BridgeVIEW | <input type="checkbox"/> VirtualBench | <input type="checkbox"/> GPIB | <input type="checkbox"/> Serial |
| <input type="checkbox"/> Lookout | <input type="checkbox"/> Measure | <input type="checkbox"/> VXI | <input type="checkbox"/> Motion control |

Tell Us About Your Applications

Number and type (AC, DC, thermocouple, and so on) of signals _____

My systems are developed by In-house staff System(s) integrator consultant

System description _____

Which statements best describe your role in the purchase of instrumentation or data acquisition products?

- I set company standards.
- I influence product purchases.
- I evaluate and recommend software.
- I use a PC regularly in my instrumentation system.
- I develop virtual instrumentation applications.

Which statement best describes your function in the company? (check only one)

- Education
- Manufacturing/automation
- Reseller/sales
- Service/repair
- Calibration
- Engineering management
- Purchasing/contracts
- Student/co-op
- Government/legal
- Research/R&D/grad student
- Software developer
- Design
- Production test
- Systems integrator/hardware
- Software consultant
- Compliance testing

Please Check Below for Free Product Information

Software Tools

- Instrupedia™/Windows (CD) – includes catalogue, software demos, application notes, and more
- Software Showcase/Windows and Macintosh (CD) – Demos of entire software line
- DAQ Designer™/Windows (3.5 in.) – DAQ system integration tool

Catalogues and Newsletters

- Measurement and Automation Catalogue*
- VXI Product Solutions Guide*
- Academic Catalogue*
- Instrumentation Newsletter™*
- Automation View™* newsletter
- Third-Party Solution CD
- NI News* e-mail newsletter

Industry-specific Literature

- Aerospace
- Telecommunications
- Automotive
- Semiconductor
- Vibration/acoustics
- Physiology
- Analytical chemistry
- Education
- Test and measurement
- Industrial automation
- Laboratory automation

Product Literature (check up to three)

- LabWindows/CVI
- ComponentWorks
- TestStand™
- LabVIEW Add-On Toolkit pack
- BridgeVIEW
- Lookout
- Analysis
- HIQ
- Measure
- LabVIEW productivity study
- DAQ
- Low-cost DAQ
- GPIB
- GPIB chip kit
- HS488™
- Virtual instrumentation software
- VXI
- VME
- PXI™
- IMAQ
- Customer education
- Computer-based instruments
- SCXI signal conditioning

Additional Literature

- NI Global Services
- Customer Education Course Schedule
- LabVIEW Technical Resource Subscription Card



11500 North Mopac Expressway Austin, TX 78759-3504
Tel: 512-794-0100, (800) 433-3488 • Fax: 512-683-9300
info@ni.com • ni.com