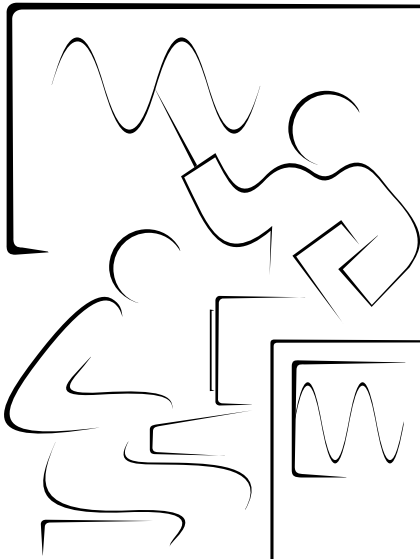




LabVIEW™ Basics II Course Manual

Course Software Version 5.1
February 1999 Edition
Part Number 320629F-01

Copyright

Copyright © 1993, 1999 by National Instruments Corporation, 6504 Bridge Point Parkway, Austin, Texas 78730-5039.
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™ is a trademark of National Instruments Corporation.
Product and company names listed are trademarks or trade names of their respective companies.

Internet Support

E-mail: support@natinst.com

FTP Site: <ftp.natinst.com>

Web Address: <http://www.natinst.com>

Bulletin Board Support

BBS United States: 512 794 5422

BBS United Kingdom: 01635 551422

BBS France: 01 48 65 15 59

Fax-on-Demand Support

512 418 1111

Telephone Support (USA)

Tel: 512 795 8248

Fax: 512 794 5678

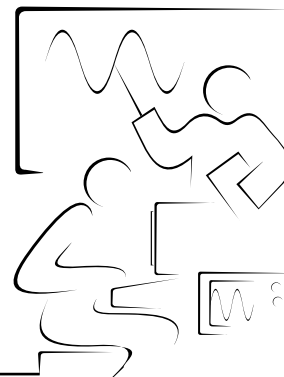
International Offices

Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 288 3336, Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 09 725 725 11, France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186, Israel 03 6120092, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico 5 520 2635, Netherlands 0348 433466, Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200, United Kingdom 01635 523545

National Instruments Corporate Headquarters

11500 North MoPac Expressway Austin, TX 78759-3504 USA Tel: 512 683 0100

Contents



Student Guide

A. Self-Paced Use.....	SG-2
B. Course Description	SG-6
C. Prerequisites.....	SG-6
D. Course Goals and Non-Goals	SG-6
E. Course Map.....	SG-8
F. Course Conventions.....	SG-9

Lesson 1

Planning LabVIEW Applications

A. The Planning and Design Process.....	1-2
B. The Implementation Process.....	1-3
Summary, Tips, and Tricks.....	1-6

Lesson 2

Clusters

A. Clusters	2-2
B. Cluster Functions	2-7
C. Error Clusters.....	2-19
D. Cluster Conversion	2-21
E. Using Polymorphism with Clusters	2-33
Summary, Tips, and Tricks.....	2-36

Lesson 3

Local and Global Variables

A. Local Variables	3-2
B. Global Variables	3-13
C. Important Advice about Local and Global Variables	3-22
Summary, Tips, and Tricks.....	3-24

Lesson 4**Attribute Nodes**

A. Attribute Nodes.....	4-2
B. Common Attributes.....	4-6
C. Graph Attributes	4-17
Summary, Tips, and Tricks.....	4-26

Lesson 5**Advanced File I/O Techniques**

A. Working with Byte Stream Files	5-2
B. LabVIEW Datalog Files	5-12
C. Streaming Data to Disk.....	5-19
Summary, Tips, and Tricks.....	5-20

Lesson 6**Developing Larger Projects in LabVIEW**

A. Assembling a LabVIEW Application	6-2
B. LabVIEW Features for Project Development.....	6-14
C. LabVIEW Add-on Packages for Project Management (Optional)	6-22
Summary, Tips, and Tricks.....	6-31

Lesson 7**Performance Issues**

A. LabVIEW Multithreading and Multitasking Overview.....	7-2
B. The VI Profile Window	7-5
C. Speeding Up Your VIs.....	7-7
D. System Memory Issues	7-19
E. Optimizing VI Memory Use	7-22
Summary, Tips, and Tricks.....	7-37

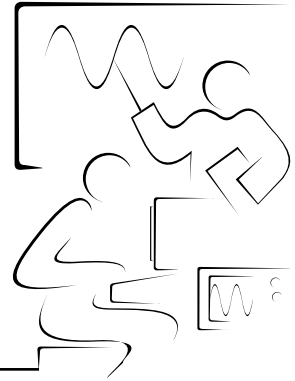
Lesson 8**Additional Topics**

A. Data Manipulation Techniques.....	8-2
B. Custom Graphics in LabVIEW.....	8-14
C. LabVIEW Run-Time Menus	8-24
D. Intensity Plots	8-33
E. Occurrences	8-36
F. Reentrant VIs	8-39
G. The CVI Function Panel Converter	8-45
H. Using LabVIEW with HiQ	8-48
Summary, Tips, and Tricks.....	8-53

Appendix

A. ASCII Character Code Equivalents Table	A-2
B. Additional Information	A-5

Student Guide



Introduction

LabVIEW (Laboratory Virtual Instrument Engineering Workbench) is a powerful instrumentation and analysis programming language for PCs running Microsoft Windows, Sun SPARCstations, Apple Macintosh computers, HP-UX workstations, and concurrent PowerMax. LabVIEW features a graphical programming environment and all the tools needed for data acquisition, analysis, and presentation. With the graphical programming language, called “G,” you can program in a block diagram notation, the natural design notation of scientists and engineers. After you create a block diagram program, LabVIEW compiles it into machine code.

LabVIEW integrates data acquisition, analysis, and presentation in one system. For acquiring data and controlling instruments, LabVIEW supports RS-232/422, IEEE 488 (GPIB), and VXI, including Virtual Instrument Software Architecture (VISA) functions, as well as plug-in data acquisition (DAQ) boards. An instrument library with drivers for hundreds of popular instruments simplifies instrument control applications. For analyzing data, the extensive Analysis library contains functions for signal generation, signal processing, filters, windows, statistics, regression, linear algebra, and array arithmetic. Because LabVIEW is graphical in nature, it is inherently a data presentation package. LabVIEW can generate charts, graphs, and custom graphics.

This guide describes the course contents and suggests ways to most effectively use the course materials. It discusses the following topics:

- A. Self-Paced Use
- B. Course Description
- C. Prerequisites
- D. Course Goals and Non-Goals
- E. Course Map
- F. Course Conventions

A. Self-Paced Use

Thank you for purchasing the LabVIEW Basics II course kit. You should be able to begin developing your application soon after you have worked through this manual. This course manual and accompanying software is used in the two-day, hands-on LabVIEW Basics II course.

To get started, read the information on the next page regarding the accompanying disks and then follow the instructions on the subsequent pages for the computer platform you are using. If you have comments, suggestions for improving this course, or are not satisfied with the material, please contact:

LabVIEW Technical Support
11500 N. MoPac Expressway
Austin, TX 78759-3504
(512) 795-8248
support@natinst.com

Attending the 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. To register for a course, or for course information, please contact National Instruments.

North America

Telephone: (512) 683-0100
E-mail: custedu.info@natinst.com
(requests for information only)
24-hour automated retrieval of course outlines and course schedules
Fax on Demand: (512) 418-1111
World Wide Web: <http://www.natinst.com>

Other Countries

Please contact your local National Instruments branch office (the phone numbers are on the second page of this manual).

Course Disk

The table below lists the contents of the LabVIEW Basics II course disk.

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

This course disk is for the Windows platform. If you do not have access to a Windows machine, contact National Instruments using the information on the previous page to get the course software for your platform.

If You Are Using LabVIEW for Windows:

Items You Need

- IBM PC AT or compatible
- LabVIEW for Windows Professional Development System, ver. 5.1 or later
- Optional—A word processing application such as NotePad or Wordpad
- HiQ version 4.0 or later

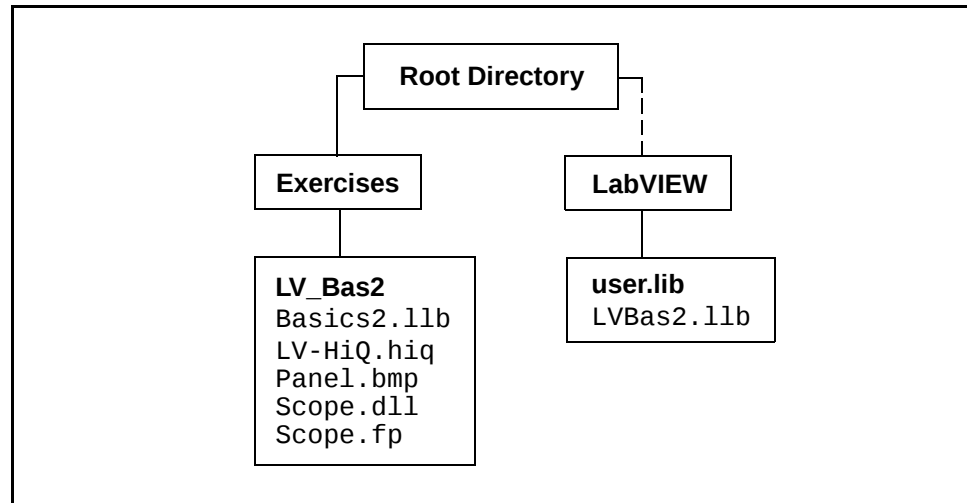
Installing the Course Software

1. Run the program called LVB2SW.exe. The following files will be extracted to the C:\Exercises\LV_Bas2 directory:
 Basics2.llb
 LV-HiQ.hiq
 Panel.bmp
 Scope.dll
 Scope.fp

LVBas2.llb will be installed in the LabVIEW\user.lib directory. When you launch LabVIEW, a palette called **Basics-II Course** will be in the **User Libraries** palette of the **Functions** palette.

2. (Optional) To install the course solutions, run LVB2Sol.exe from the floppy disk. The files Bas2Soln.llb, Menu.rtm, and Scope.llb will be installed to the C:\Solutions\LV_Bas2 directory.

The course assumes the following directory structure:



If You Are Using LabVIEW for Sun:

Items You Need

- Sun SPARCstation computer or compatible running XWindows
- LabVIEW for Sun Professional Development System, ver. 5.1 or later
- Optional—A word processing application such as Text Editor

Installing the Course Software

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 shown above. Copy all files to the appropriate location.

If You Are Using LabVIEW for HP-UX:

Items You Need

- Hewlett-Packard Model 9000 Series 700 workstation running HP-UX 9.0.3 or later with an X Window System Server
- LabVIEW for HP-UX Professional Development System, ver. 5.1 or later
- Optional—A word processing application such as vi or vuedpad

Installing the Course Software

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. Log in to your workstation as a superuser and copy the files to your hard disk using the directory structure shown on the previous page.

If You Are Using LabVIEW for Macintosh:

Items You Need

- Macintosh or Power Macintosh computer
- LabVIEW for Macintosh Professional Development System, ver. 5.1 or later
- Optional—A word processing application such as TeachText

Installing the Course Software

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 shown on the previous page.

B. Course Description

The LabVIEW Basics II course teaches you to make optimum use of LabVIEW for developing your applications. The course is divided into lessons, each covering a topic or a set of topics. Each lesson consists of:

- An introduction that describes the lesson's purpose and what you will learn.
- A discussion of the topics.
- A set of exercises to reinforce the topics presented in the discussion.
- A set of additional exercises to be done if time permits.
- A summary that outlines important concepts and skills taught in the lesson.

C. Prerequisites

- Familiarity with the Macintosh, Windows, Sun, or HP-UX operating system.
- Experience writing algorithms in the form of flowcharts or block diagrams.
- LabVIEW Basics I course or equivalent experience.

D. Course Goals and Non-Goals

This course prepares you to:

- Take advantage of advanced data types.
- Use local and global variables in your applications.
- Programmatically customize user interfaces using attribute nodes.
- 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 6, *Developing Larger Projects in LabVIEW*. In Lesson 6, you will build a project that uses VIs you create in Lessons 2, 3, 4, and 5. While these VIs individually illustrate specific concepts and features in LabVIEW, they constitute part of a larger project you will finish in Lesson 6.

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.
- Provides user-configurable sample rate, number of samples, and alarm limit.
- Acquires data with specified user configuration.
- Allows the user to analyze a subset of data and save the results to a file.
- The user can load and view analysis results previously saved to disk.

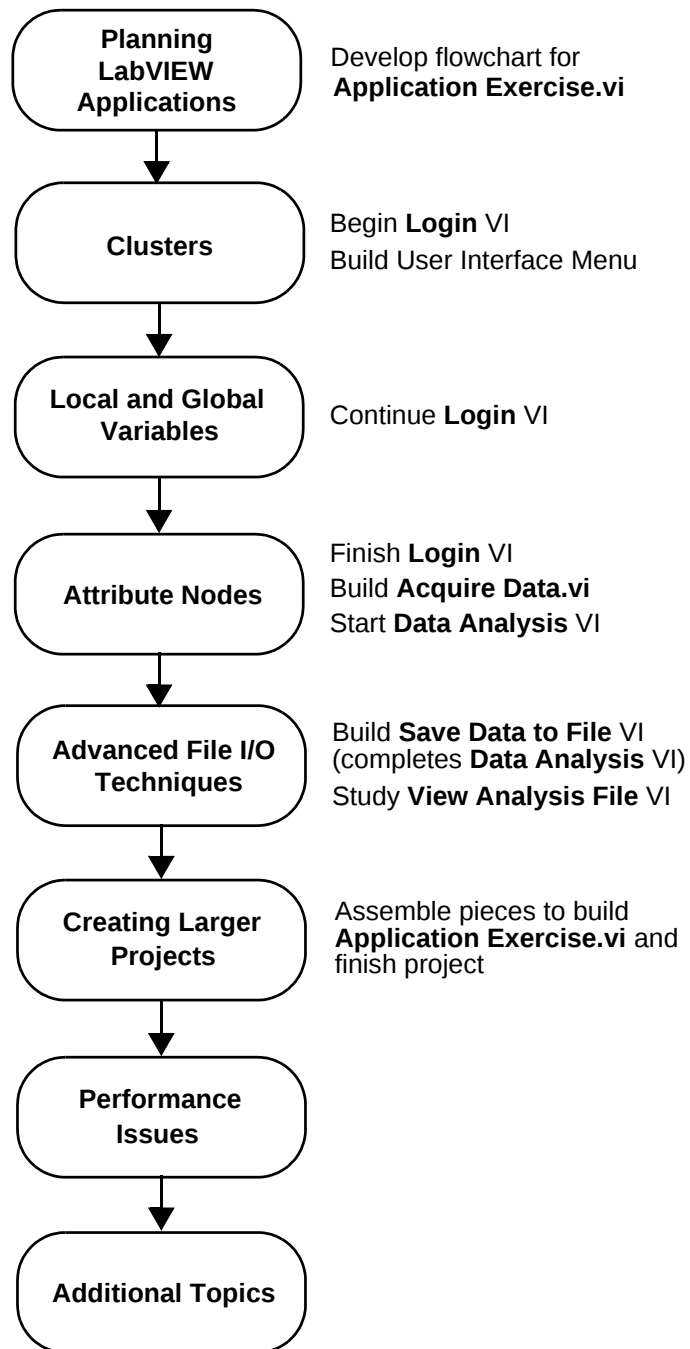
Additional optional exercises challenge you to enhance the basic application features. Specific details regarding the program capabilities are in the relevant exercises.

The course map on the following page features notes about the parts of the project you will develop in various sections of the course. Specific exercises within the chapters also remind you when you are working on a VI used in a later exercise.

It is not the purpose of the course to discuss any of the following:

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

E. Course Map



F. Course Conventions

The following conventions are used in this course manual.

Bold

Words in bold refer to LabVIEW menus, menu items, palettes, subpalettes, functions, and VIs. For example, **File**.

Italics

Words in italics are for emphasis.

Courier

Words in Courier indicate drive names, libraries, directories, pathnames, filenames, and sections of programming code. Courier also indicates information you must type. For example, type `Digital Indicator` inside the bordered box.

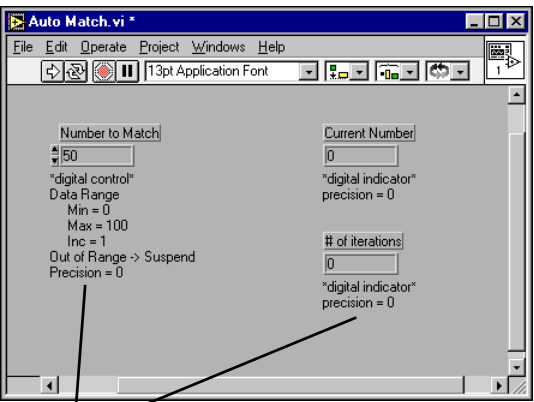
<shift>

Angle brackets enclose names of keys. In some places, keys for all four platforms are shown using the following convention:

<ctrl | ⬠ | M | option>
Win Sun H-P Mac

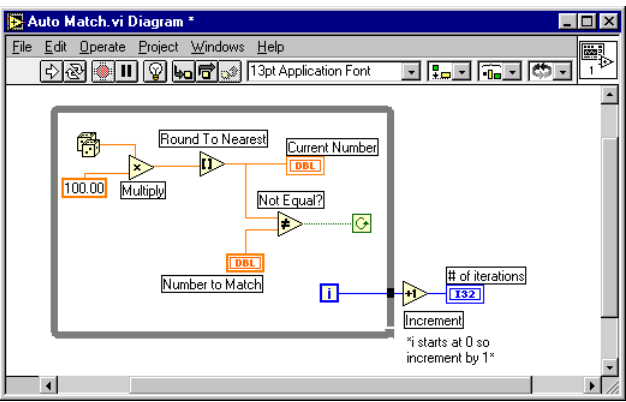
As shown below, each exercise shows a picture of a *finished* front panel and block diagram. The front panel picture shows the front panel *after* you run the VI. After each block diagram picture is a description of each object in the block diagram and where you can find the object.

Front Panel



Comments
(Do not enter these)

Block Diagram



Name of object

↓

 **Random Number (0-1) function (Arithmetic menu).**
This function returns a random number between 0 and 1.

↓

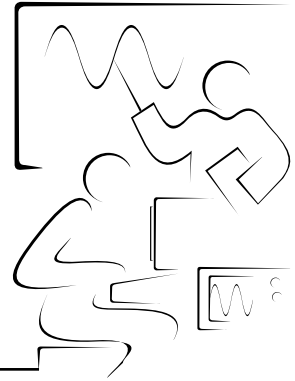
Description of object

Location of object

↓

Lesson 1

Planning LabVIEW Applications



Introduction

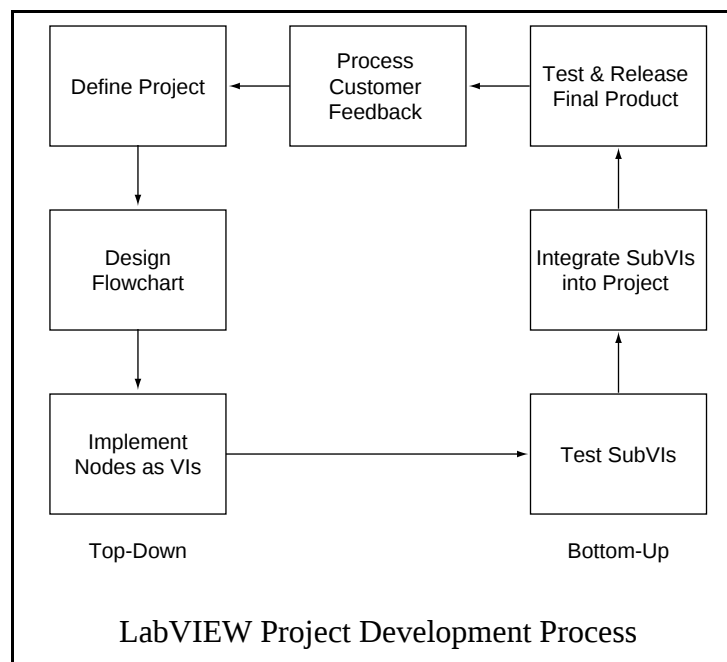
This lesson discusses some of the issues involved when developing LabVIEW applications, including the design process, the organization of subVI components, and the process of combining those components to create a complete application.

You Will Learn:

- A. Planning and design tips for developing a LabVIEW application.
- B. How to convert your design outline into actual LabVIEW subVIs.

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 dataflow programming language and its 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 diagram without subVIs, you may 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, as well as 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.

Exercise 1-1

Objective: To develop a flowchart describing a generic data acquisition application.

In the following lessons, you will develop a generic data acquisition application 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, and alarm limit.
- Acquires data with the specified user configuration.
- As soon as the data has been acquired, and any time the user chooses 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 press of a Quit button.

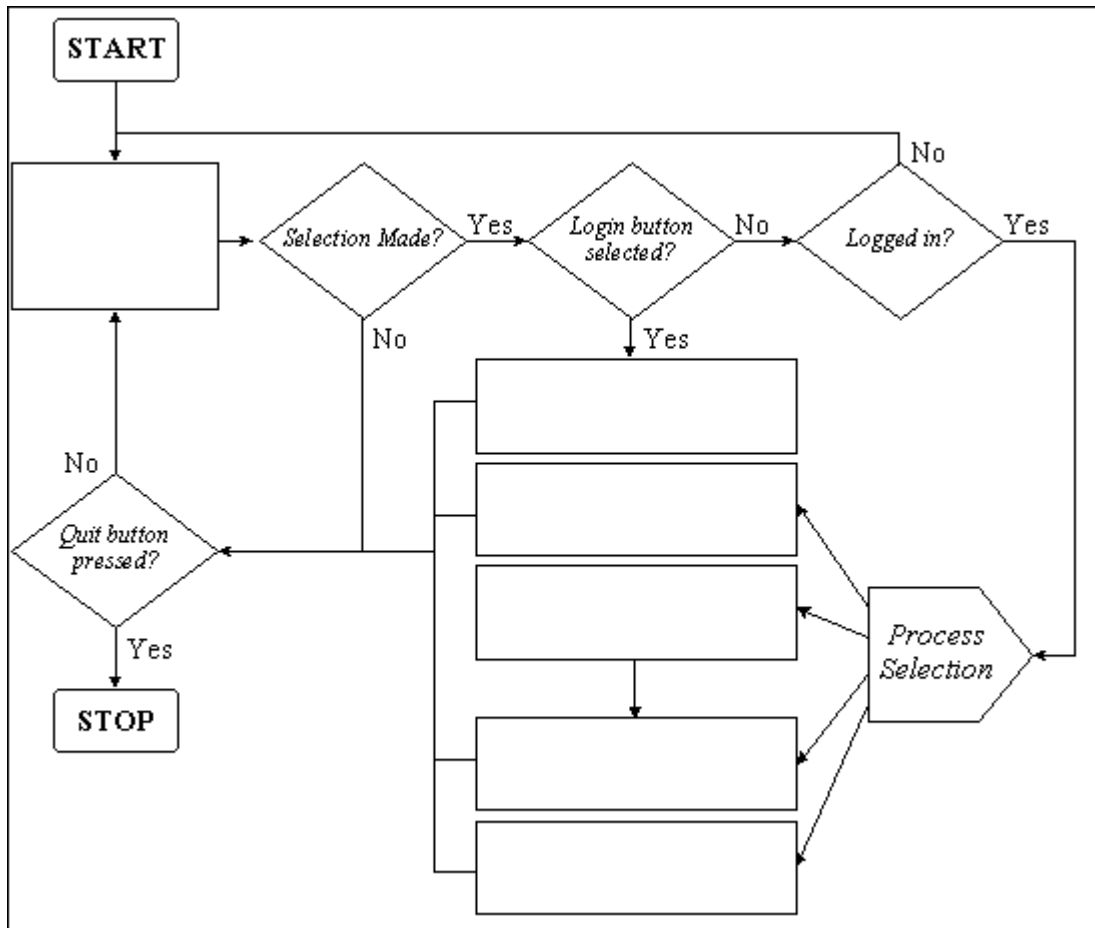
Part A—Detailing the Application

While you have a general overview of the application's features, there are many details that remain unclear. In the space below, write down some questions that you need to ask to better define the criteria of this application:

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.
- 7.
- 8.
- 9.
- 10.

Part B—Developing the Application Flowchart

Using the picture below, complete a flowchart describing the application based on the above criteria. The program flow should begin in the rectangular box in the top-left corner of this illustration:



Each rectangular box should incorporate some feature of the overall application. In lessons 2, 3, 4, and 5, you will develop subVIs that correspond to the functions in the rectangular boxes of the flowchart. In Lesson 6, you will integrate those subVIs into an overall application following the above flow diagram.

When developing larger applications, it may prove useful to break each node of your flowchart into its own flow diagram, organizing each subVI of the application into a series of logical steps and subVIs. The flowcharting process is an iterative one and can be used at lower and lower levels until numerous small subVIs have been created.

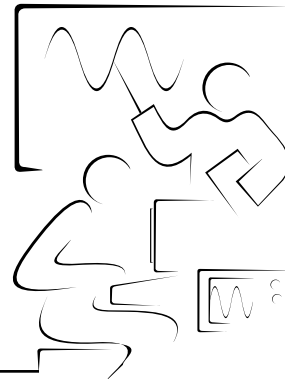
End of Exercise 1-1

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 may 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 LabVIEW VIs to accomplish the various steps in your flowchart. It is a good idea to modularize your application into logical subtasks whenever possible. By working with small modules, you can easily debug an application by testing each module individually. This approach also makes it much easier to modify the application in the future.

Lesson 2

Clusters



Introduction

When implementing tasks such as handling menu selections and verifying access information, you often encounter collections of data. For example, to verify access information, you would need to work with a collection of data that could consist of a name, password, and ID number. Collections of data that have an implicit association among their elements lend themselves well to data structures such as structs in C or records in Pascal. In LabVIEW, you can store such data as clusters and subject it to powerful manipulation using cluster functions.

You Will Learn:

- A. What clusters are.
- B. Some cluster functions.
- C. How to use error clusters.
- D. How to convert clusters to one-dimensional (1D) arrays and vice-versa.
- E. How to use polymorphism with clusters.

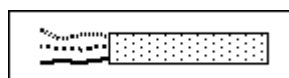
Application SubVIs You Will Build:

- A. **Verify Information VI**
- B. **Check Employee VI**
- C. **Menu VI**

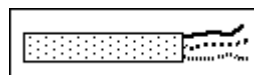
A. Clusters

A cluster is a data structure that combines one or more data components into a new data type. The components that form a cluster may have different data types such as Boolean, string, and integer data types. A cluster is analogous to a *record* in Pascal or a *struct* in C.

On the block diagram, a wire that carries the data stored in a cluster may be thought of as a bundle of smaller wires, much like a telephone cable. Each wire in the cable represents a different component of the cluster. Because a cluster constitutes only one “wire” in the block diagram, clusters reduce wire clutter and the number of connector terminals that subVIs need.

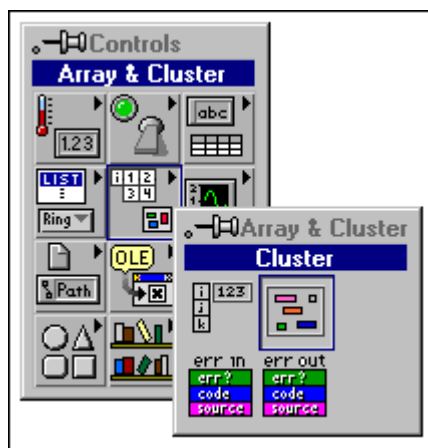


You “unbundle” the cluster in the diagram to access its components. You may think of unbundling a cluster as unwrapping a telephone cable and accessing the individual wires within the cable.

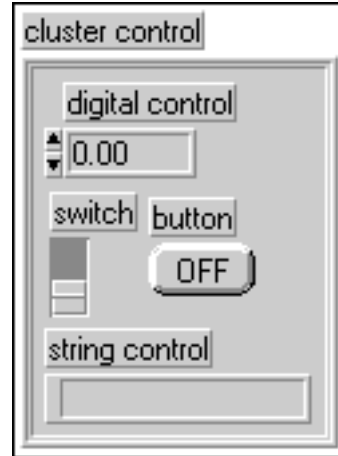


Creating Cluster Controls and Indicators On the Front Panel

You create cluster controls and indicators on a VI’s front panel by placing a *cluster shell* in the Panel window. To place an empty cluster shell on the panel, choose **Array & Cluster»Cluster** from the **Controls** palette. Then, click in the Panel window to place the cluster. You can adjust the size of the cluster shell by holding down the mouse button and dragging the cursor when placing the cluster shell on the panel.

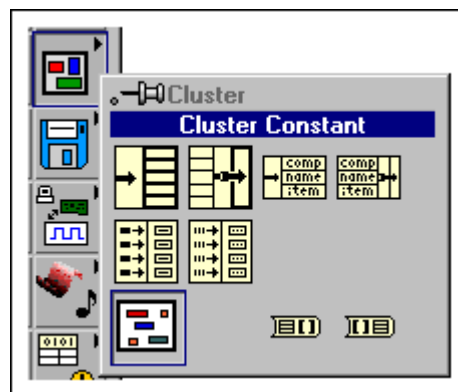


You can place any objects inside the cluster that you normally place in the Panel window. You can deposit objects directly inside the cluster by dragging an object into a cluster. *Objects inside a cluster must be all controls or all indicators.* You cannot combine both controls and indicators inside the same cluster. For example, if you drop an indicator into a cluster containing controls, the indicator changes to a control. The cluster assumes the data direction (control or indicator) of the first object you place inside the cluster. A cluster with four controls is shown below.



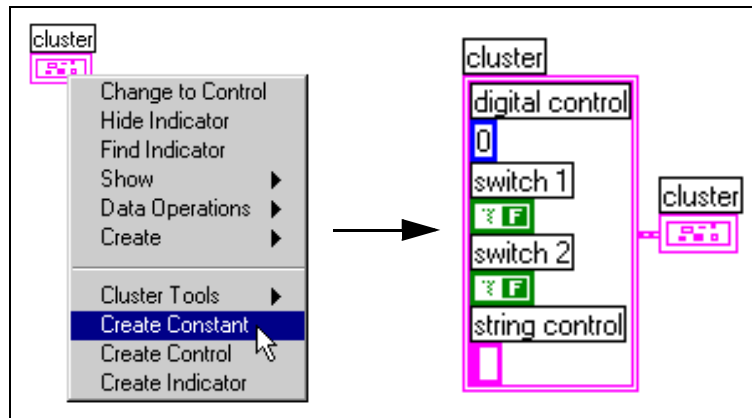
Creating Cluster Constants on the Block Diagram

To create a cluster constant on the block diagram, you can use the same technique as you used on the front panel. From the Diagram window, choose **Cluster»Cluster Constant** from the **Functions** palette to create the cluster shell. Then, place other constants of the appropriate data type within the cluster shell.



If you have a cluster control or indicator on the front panel, and would like to create a cluster constant containing the same components in the diagram, you can pop up on its terminal and select **Create Constant** from the pop-up

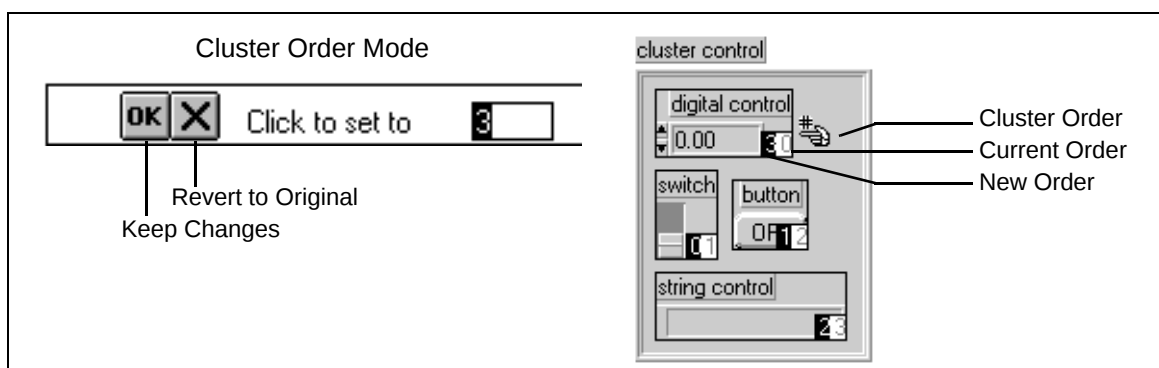
menu. This technique is a great time-saver. If you use this method on a cluster indicator, the constant is wired to the indicator automatically.



Cluster Order

When LabVIEW manipulates clusters of data, not only are the data types of the individual components within the cluster important, but also the order of the components in the cluster. Cluster components have a logical order unrelated to their position within the shell. The first object placed in the cluster shell is component 0, the second is component 1, and so on. If you delete a component, the order adjusts automatically.

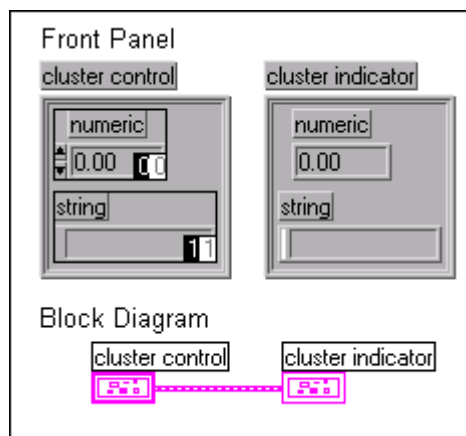
You can change the order of the objects within the cluster by popping up on the cluster *border* and choosing **Cluster Order...** from the pop-up menu. A new set of buttons replaces the toolbar, and the cluster appearance changes as shown below. The white box on each component shows its current place in the cluster order. The black box shows a component's new place in the order.



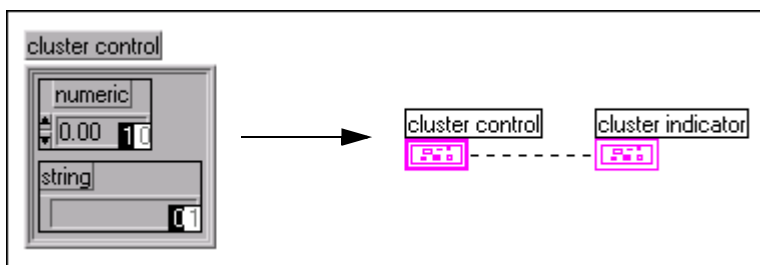


To set a cluster component to a particular index in the cluster order, first type the desired order number into the “Click to set to” field. Then click on the desired component. You will notice that the component’s cluster order index changes. You will also notice that the cluster order indices of the other components adjust automatically. To save your changes, click on the OK button in the palette. To revert to the original settings, click on the Revert to Original button. You use this technique to set the cluster order for constants on both the front panel and the block diagram.

The example shown below illustrates the importance of cluster order. The front panel contains two simple clusters. In the first cluster, component 0 is a numeric control, and component 1 is a string control. In the second cluster, component 0 is a numeric indicator, and component 1 is a string indicator. The cluster control wires to the cluster indicator on the block diagram.

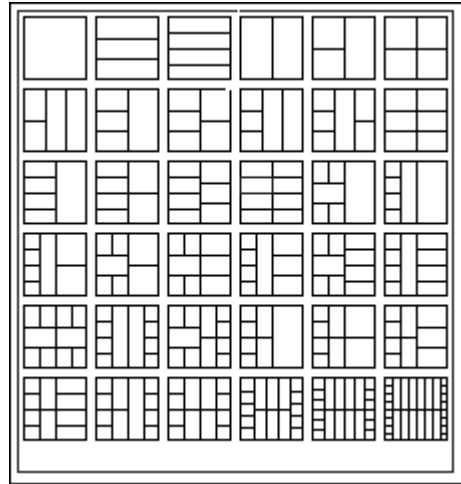


However, if you change the cluster order of the indicator so the string indicator is component 0 and the numeric is component 1, the wire connecting the control to the indicator is broken. If you try to run the VI, you get an error message stating that there is a type conflict because the data types do not match.

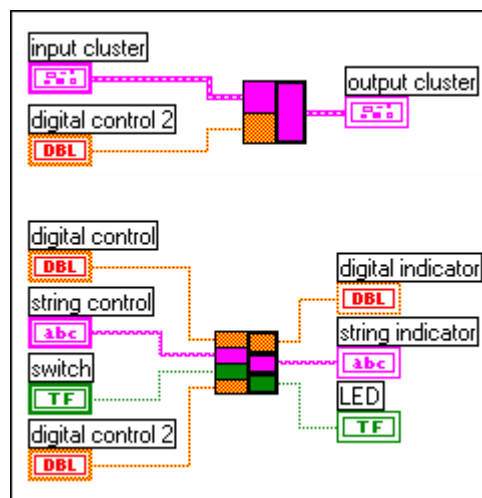


Using Clusters to Pass Data to and from SubVIs

As shown in the figure below, the connector pane of a VI can have a maximum of 28 terminals. When you use a connector pane that has a large number of terminals, the terminals are very small. With such small terminals, wiring errors are more likely.



By bundling a number of controls into a cluster, and passing the cluster to the subVI, you can overcome the 28-terminal limit of the connector pane. One cluster control uses one terminal on the connector pane, but that cluster can contain several controls. Similarly, one terminal assigned to a cluster indicator can pass several outputs from the subVI. Because your subVI uses clusters containing several items each, you can use fewer, and therefore larger, terminals on the connector pane. This will make for cleaner wiring on the diagram.



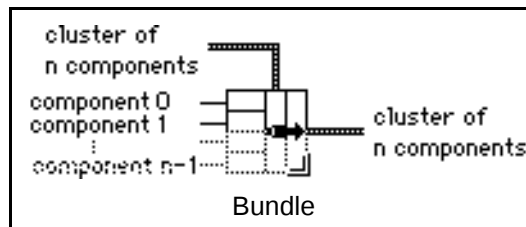
B. Cluster Functions

LabVIEW uses several functions to manipulate clusters. We will study the **Bundle** and **Bundle by Name** functions, which assemble and modify clusters, and the **Unbundle** and **Unbundle by Name** functions, which disassemble clusters.

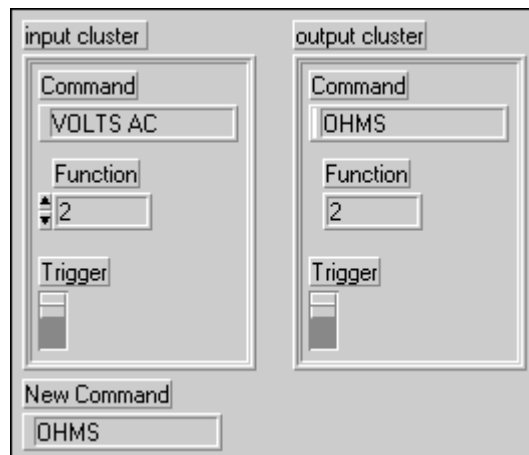
Assembling Clusters



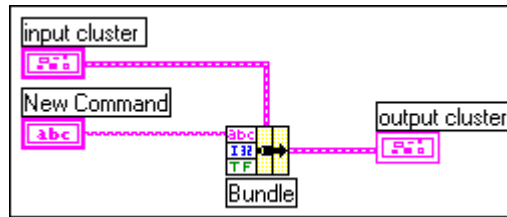
The **Bundle** function (**Cluster** palette) assembles individual components into a single cluster or replaces components within an existing cluster. The topmost component wired to the **Bundle** function is component 0 in the cluster; the component beneath it is component 1; and so on. You can increase the number of inputs by resizing the function with the Positioning tool or by popping up on the icon and selecting **Add Input** from the pop-up menu. If the **cluster of n components** terminal is wired, the number of input terminals to the **Bundle** function must match the number of items in the input cluster.



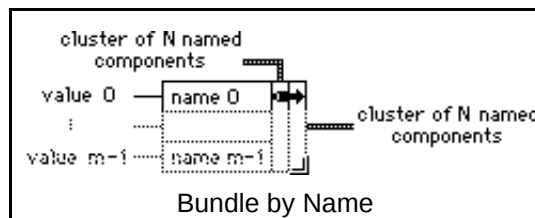
When you use the **cluster of n components** terminal, you do not need to wire data to every input terminal of the function. Instead, you can wire data only to the items you want to change. For example, consider the cluster shown below, which contains three controls—a string labeled “Command,” a numeric labeled “Function,” and a Boolean labeled “Trigger.”



You can use the **Bundle** function to change the string value by wiring the components as shown below. You must know the cluster order to do this correctly.

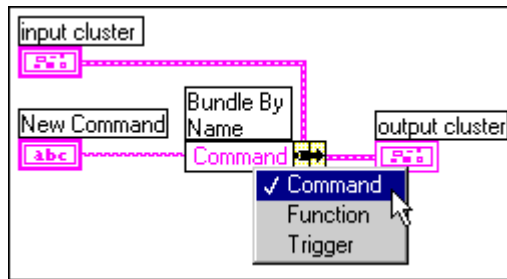


The **Bundle by Name** function replaces components in an existing cluster. **Bundle by Name** works similarly to the **Bundle** function, but instead of referencing cluster components by their cluster order, it references them by their owned labels. You cannot access components in the input cluster (connected to the **cluster of N named components** input terminal) that do not have owned labels.

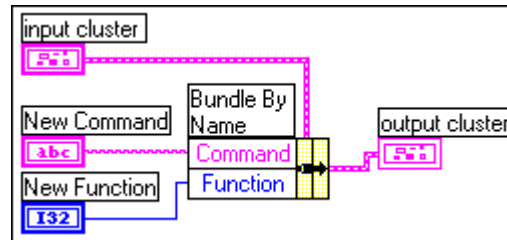


You select a component by clicking on an input terminal using the Operating tool and selecting a name from the list of components in the cluster. Also, you can pop up on the input terminal and select the component from the **Select Item** menu. Note that you *must* wire an input cluster to the **cluster of N named components** input of this function, and at least one item in the input cluster must have a name. The number of terminals on the **Bundle by Name** icon does not need to match the number of components in the input cluster.

For example, consider again the cluster control containing the “Command,” “Function,” and “Trigger” controls. You can use the **Bundle by Name** function to change the string value by wiring the components as shown in the next figure. To select the name “Command,” you click on the left terminal of the **Bundle by Name** function using the Operating tool, and select “Command” from the list of names.



If you need to modify both “Command” and “Function,” you can resize the **Bundle by Name** function as shown below.

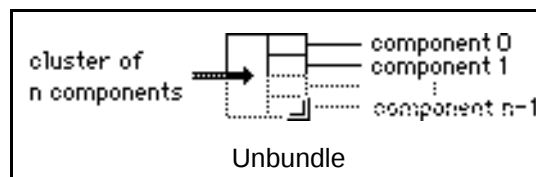


The **Bundle by Name** function is useful when working with data structures that may change during the development process. If you add a new component to the cluster or modify its order, you do not need to rewire the **Bundle by Name** function on the diagram because the names are still valid.

Disassembling Clusters

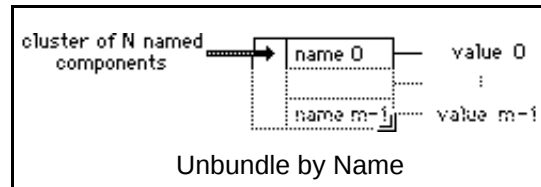


The **Unbundle** function (**Cluster** palette) splits a cluster into each of its individual components. The components are arranged from top to bottom according to the cluster order of the input cluster. You can increase the number of outputs by resizing the function with the Positioning tool or by using the pop-up menu. The number of output terminals for this function must match the number of components in the input cluster.

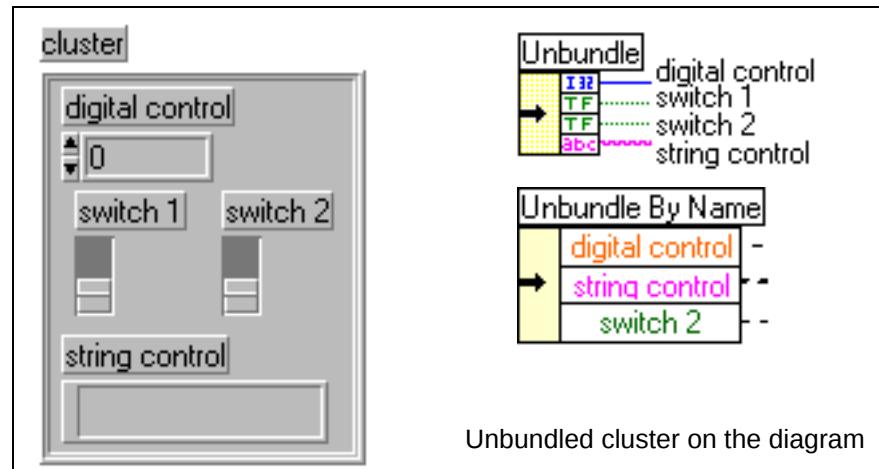


The **Unbundle by Name** function (**Cluster** palette) returns the cluster components that you reference by name. You select a component by clicking on the output terminal using the Operating tool and selecting a name from the list of components in the cluster. Also, you can pop up on an output terminal and select the component from the **Select Item** menu. Because the cluster components are referenced by name, you can access only the cluster components that have owned labels. The number of output terminals of the

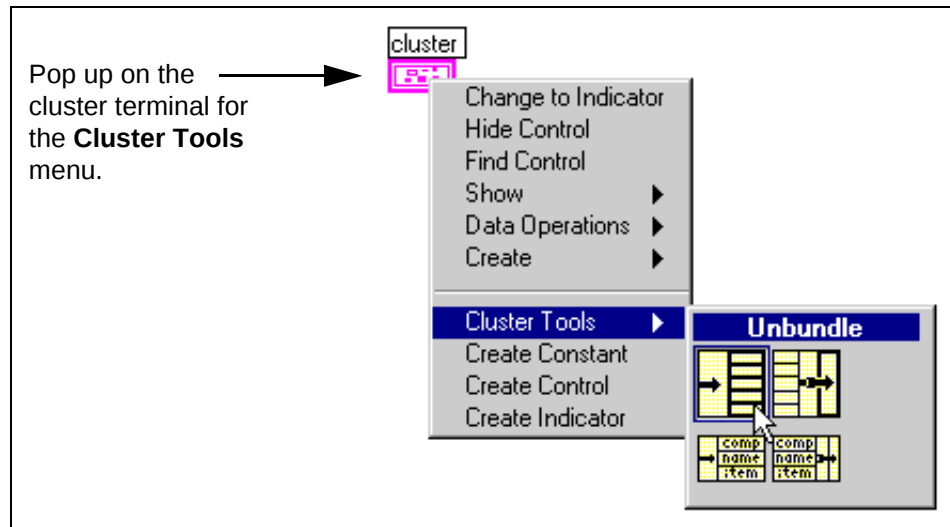
Unbundle by Name function does not depend on how many components are in the input cluster.



For example, if you used the **Unbundle** function with the cluster shown below, it would have four output terminals. These four terminals correspond to the four controls inside the cluster. Notice that you need to know the cluster order so that you can associate the correct Boolean (TF) terminal of the unbundled cluster with the corresponding switch inside the cluster. In this example, the components are ordered from top to bottom starting with component 0. Notice that when you use the **Unbundle by Name** function, you can have an arbitrary number of terminals and access specific components by name in any order.



As shown below, you can also create the **Bundle**, **Bundle by Name**, **Unbundle**, and **Unbundle by Name** functions by popping up on a cluster terminal in the block diagram and choosing **Cluster Tools** from the pop-up menu. The **Bundle** and **Unbundle** functions will automatically contain the correct number of terminals. The **Bundle by Name** and **Unbundle by Name** functions will appear with the first component in the cluster.



Exercise 2-1 Verify Information.vi

Objective: To use clusters and cluster functions.

This VI verifies that an input name and password matches one of the records stored in an array of employee records. You will use this VI as a subVI in subsequent exercises to verify more information.



Note

You will use this VI in the project in Lesson 6.

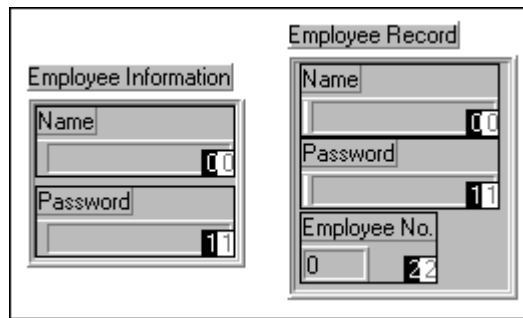
Front Panel



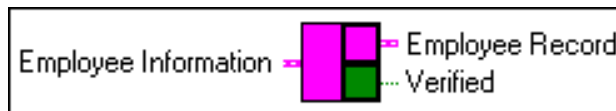
1. Open the **Verify Information** VI in C:\Exercises\LV_Bas2\Basics2.11b. The block diagram of this VI is partially built. You will build the front panel and finish the rest of the diagram.
2. Create the Employee Information cluster control. This cluster contains a name and password that will be tested for a match in the Employee Records.
 - a. Place a cluster shell in the panel by choosing **Cluster** from the **Array & Cluster** palette. Label it *Employee Information*.
 - b. Place two string controls inside the cluster, labeling the first one *Name* and the second *Password*. Pop up on each string control and select the **Limit to Single Line** option. If you need to enlarge the shell, drag one of the corners with the Positioning tool.
3. Create the Employee Record cluster indicator. This cluster contains the login name, password, and employee number when a match is found for the Employee Information cluster.
 - a. Place another cluster shell in the panel and label it *Employee Record*.
 - b. Place two string indicators inside the cluster shell, labeling the first one *Name* and the second *Password*. Then place a digital indicator inside the cluster and label it *Employee No.* To change the representation of the digital indicator to **long** (I32), pop up on

the indicator and select **Representation** from the pop-up menu. Recall that the cluster becomes an indicator if the first item you place inside of it is an indicator.

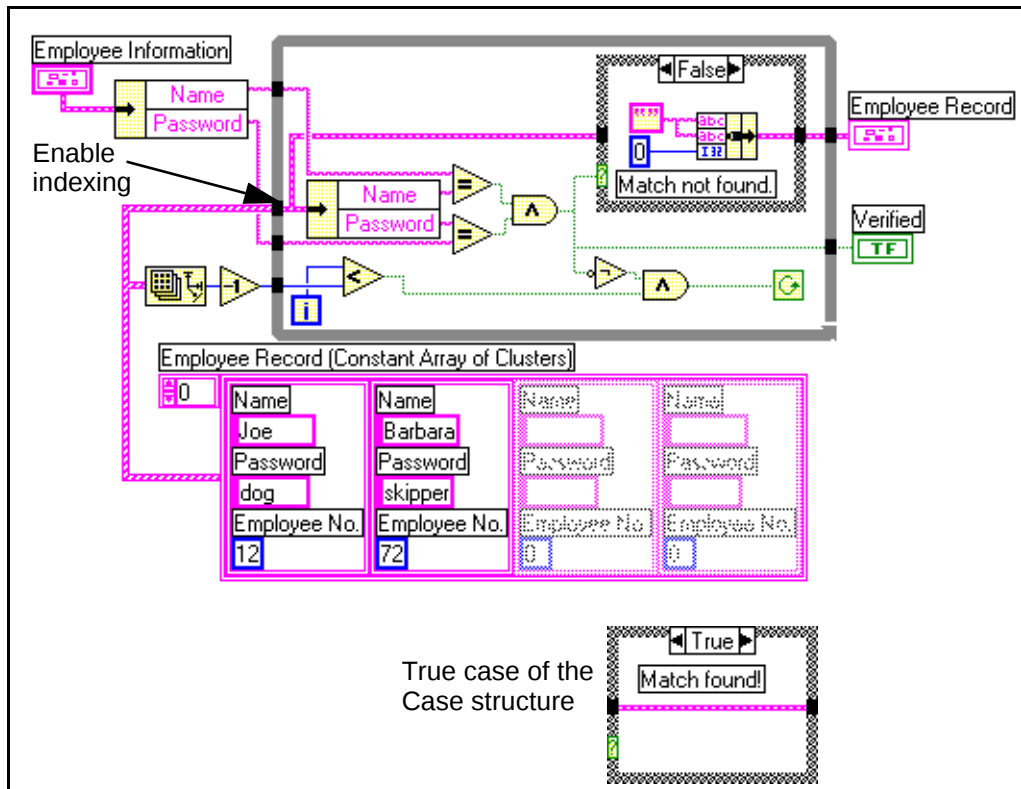
4. Verify the cluster order of the components in the Employee Information and Employee Record clusters. Pop up on the boundary of each cluster and choose **Cluster Order** from the pop-up menu. Make sure the cluster order matches the ones shown below.



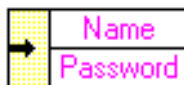
5. Create a round LED indicator (**Boolean** palette) and label it *Verified*.
6. Show the VI's connector pane by selecting **Show Connector**. Create the connector pane as shown below.



Block Diagram



1. Finish building the block diagram shown above. The **Unbundle by Name** functions compare the contents of the Employee Information cluster and the Name and Password components of an employee record. The **Bundle** function creates a default value for the Employee Record output cluster if a match cannot be found.



Unbundle by Name function. Pop up on the Employee Information terminal and choose **Cluster Tools»Unbundle by Name** from the pop-up menu. Increase the number of outputs to two. Ensure that the correct outputs are selected by clicking with the Operating tool on the top right terminal, and selecting **Name** to unbundle the Name control. Repeat this process on the lower right terminal, choosing **Password** to unbundle the Password control. Copy this icon for later use and place it inside the While Loop.



Bundle function. Pop up on the Employee Record terminal and choose **Cluster Tools»Bundle** from the pop-up menu. Place the icon inside the False case of the Case structure inside the While Loop.



Empty String constant (**String** palette). Connect this constant to the top two input terminals of the **Bundle** function. Wire a numeric constant of zero to the remaining input terminal.

2. Create the array constant containing employee records to compare with information in the Employee Information cluster.
 - a. Create an empty array constant by selecting **Array»Array Constant** from the **Functions** palette.
 - b. Pop up on the Employee Record terminal and choose **Create Constant** from the pop-up menu. Delete the wire that connects the resulting constant to the indicator. (This wire will not appear if you already wired an object to the Employee Record terminal before selecting **Create Constant**.)
 - c. Drag the cluster constant into the empty array constant. Enlarge the array constant so that several of the elements are visible. By typing information into the elements of the array constant, you create the list of employee records to search.

**Note**

To simplify moving and resizing elements in the cluster constant, pop up on the cluster constant border and deselect Autosizing. To show the owned labels of each object in the cluster constant, pop up on the object in the constant and select Show»Label.

- d. Connect this constant to the **Array Size** icon and also to the **Unbundle by Name** icon you placed inside the While Loop in step 1.

**Note**

If you get a broken wire when connecting the array constant to the Unbundle by Name icon inside the While Loop, pop up on the tunnel on the While Loop border and select Enable Indexing.

3. Return to the front panel and run the VI with various name and password combinations in the Employee Information cluster. The While Loop will stop as soon as it finds a match, or when it reaches the end of the array, whichever happens first. Verify that the VI operates correctly.
4. Save and close the VI.

**Note**

Save all your VIs in Basics2.llb.

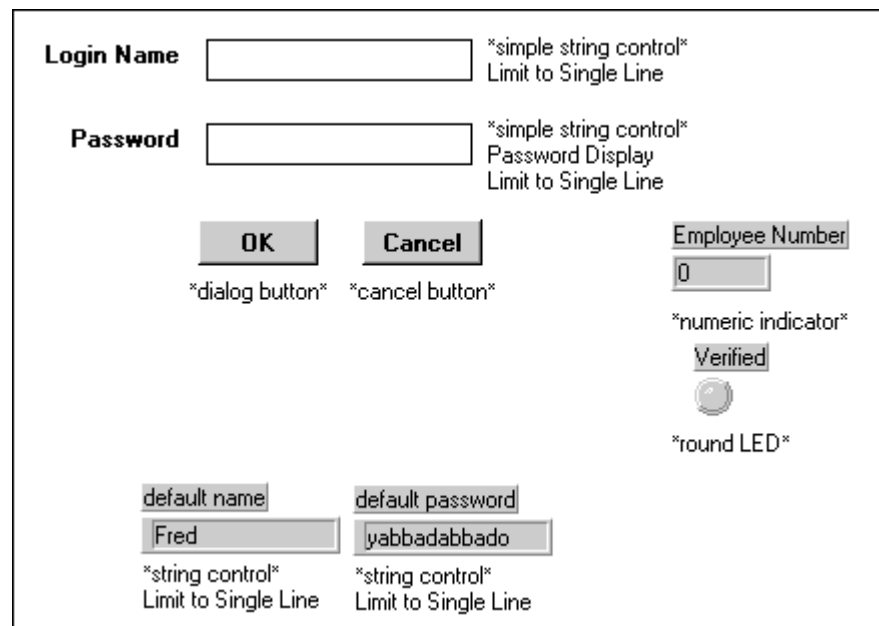
End of Exercise 2-1

Exercise 2-2 (Optional) Check Employee.vi

Objective: To use clusters to pass data into and out of a subVI.

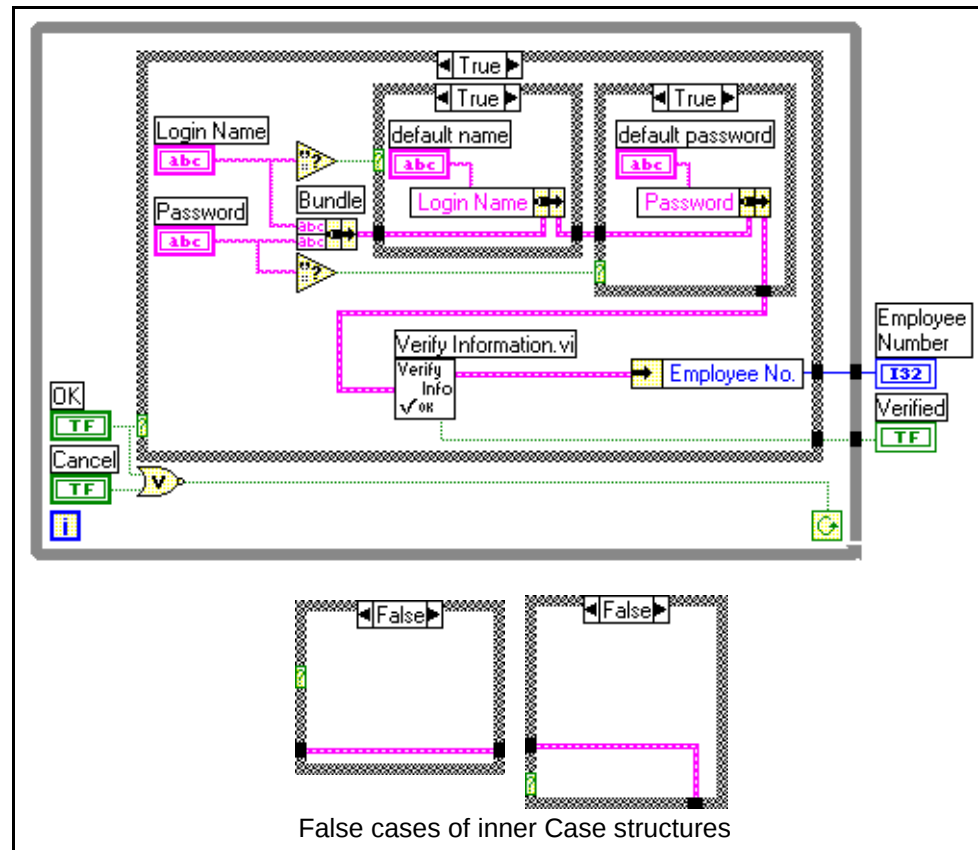
You will complete a VI that allows you to enter an employee name and password, and then searches for a match in an array of records using the **Verify Information** VI from Exercise 2-1. If a match is located, the Verified LED is turned on and the employee number retrieved from the employee record is displayed in a digital indicator. If the user does not type in a name or password, a default name or password is sent to the **Verify Information** subVI.

Front Panel



1. Open the **Check Employee** VI in Basics2.11b. The front panel of this VI has already been built. Descriptions of the controls and indicators are in the online help. Select **Help»Show Help** to show the Help window and read the descriptions of the controls and indicators.

Block Diagram



1. Study the incomplete block diagram. The large False case inside the While Loop has already been built. You will complete the rest of the diagram.
2. Create a **Bundle** function to bundle the Name and Password controls together. Then, create the **Bundle by Name** functions to use in the two Case structures. If the Login Name control is empty, replace the contents of the cluster using the **Bundle by Name** function. Then check to see if the Password control is empty. If so, substitute the default password into the cluster.



Note

*Recall that you cannot select the name of the input item in a **Bundle by Name** function until you have connected a cluster to the center input terminal of the icon.*

3. Place the **Verify Information** VI, which you created in Exercise 2-1, in the Case structure. After checking the contents of the cluster containing Login Name and Password, the VI should send the cluster to the Employee Information input of the **Verify Information** subVI. Use **Unbundle by Name** to get Employee No.



Note

If you did not complete Exercise 2-1, you can find the Verify Information VI in Bas2Soln.llb. The VI library containing exercise solutions is stored in a folder called Solutions.

4. Save the VI. Run it to test several alternatives. Press the **OK** button on the front panel to verify the name and password.
5. Close the VI when you are finished.

End of Exercise 2-2

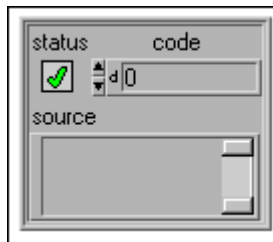
C. Error Clusters

When designing applications, you should try to anticipate any errors that may occur. For example, if you acquire signals and save them to a file over a long period of time, the operation might fail at some point due to a lack of disk space. If your application requires reliable operation, you should check for errors and decide what to do when an error occurs.

LabVIEW I/O operations check for errors and return error codes. Because error handling is different for every application, LabVIEW does not perform automatic handling of all errors. For example, you may want to terminate an application if an error occurs, or you may just want to notify the user so that he or she can correct the problem.

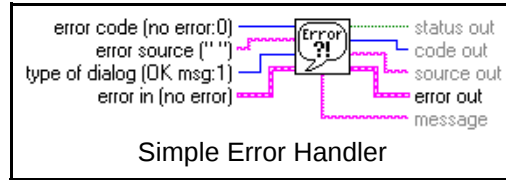
Most I/O operations in LabVIEW incorporate an error cluster that makes error handling easier. Under this scheme, I/O VIs have both an error input and an error output. If the input contains an error, the I/O VI either does nothing or shuts down the I/O operation it manages. One advantage of this approach is that you can connect several I/O operations together so that, if an error occurs, LabVIEW will not perform subsequent I/O operations.

The error cluster used by most of the I/O operations—including data acquisition (DAQ), DDE, TCP, UDP, VISA, GPIB, and the File I/O functions—contains a Boolean indicating whether an error has been detected, a numeric error code, and a string used to identify the source of the error.

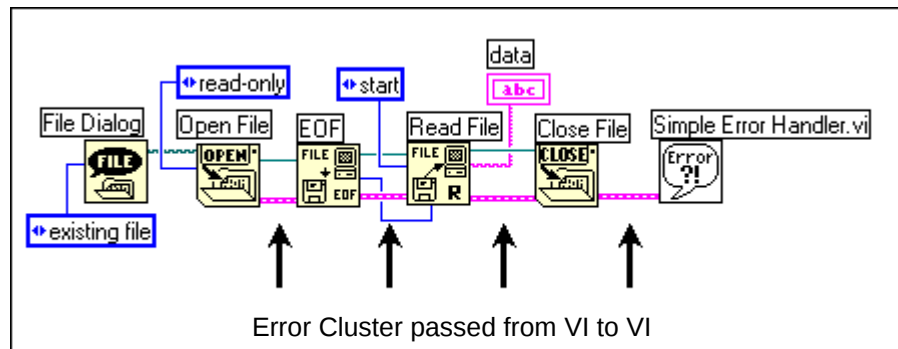


To make error handling easier for you, LabVIEW features several error handling functions. The **Simple Error Handler** VI takes care of the error handling that most applications need. This handler contains error information about all I/O operations in LabVIEW.

The **Simple Error Handler** VI (Time & Dialog palette) determines whether an error has occurred. If it finds an error, this VI creates a description of the error. You can then set the VI to display the error description in a dialog box.



Consider the following example. It uses the error cluster scheme to pass errors from one VI to the next. Thus, if an error occurs in **Open File.vi**, the error passes through and into **EOF.vi**. The **EOF** VI detects the error and, as such, does not perform its normal operations. Instead, it passes the unchanged error information out to **Read File.vi**. Likewise, the **Read File** VI will detect the error and pass the error information. The error cluster finally reaches the **Simple Error Handler** VI, which displays a dialog box with the error information.



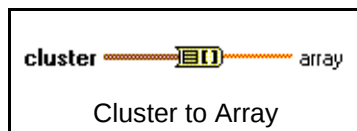
The above example illustrates a typical usage of the *error in/error out* approach. That is, the File I/O VIs use error clusters to pass information from one operation to the next.

To incorporate error handling in your own VIs (which we recommend), you can use the front panel error clusters in **Controls»Array & Cluster**. From this palette, you can easily place **error in** and **error out** clusters on your front panel.

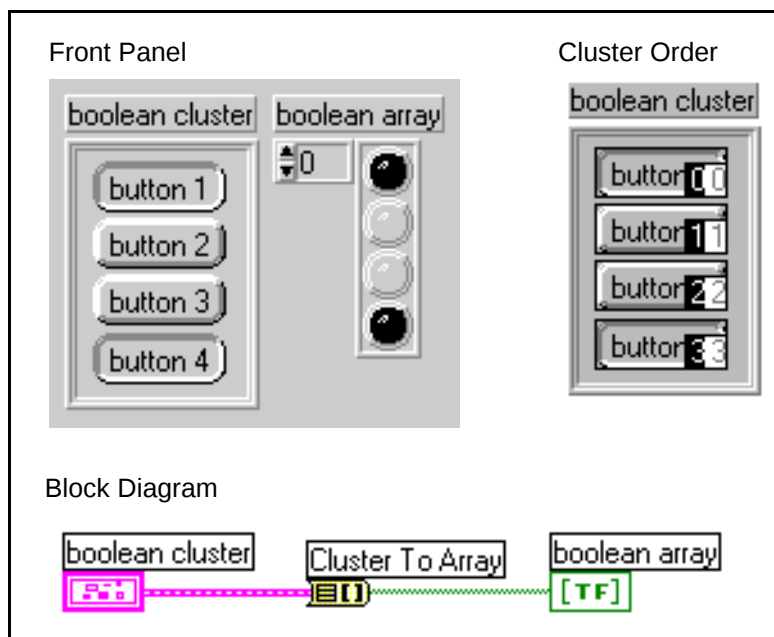
D. Cluster Conversion

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

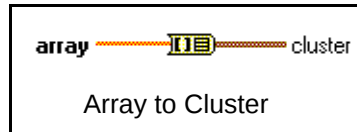
The **Cluster to Array** function (**Cluster** and **Array** palettes) converts a cluster of identically typed components to a 1D array of the same data type.



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

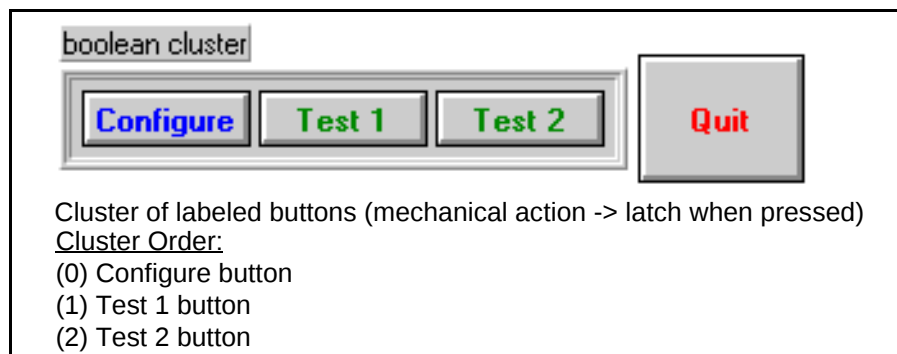


Note

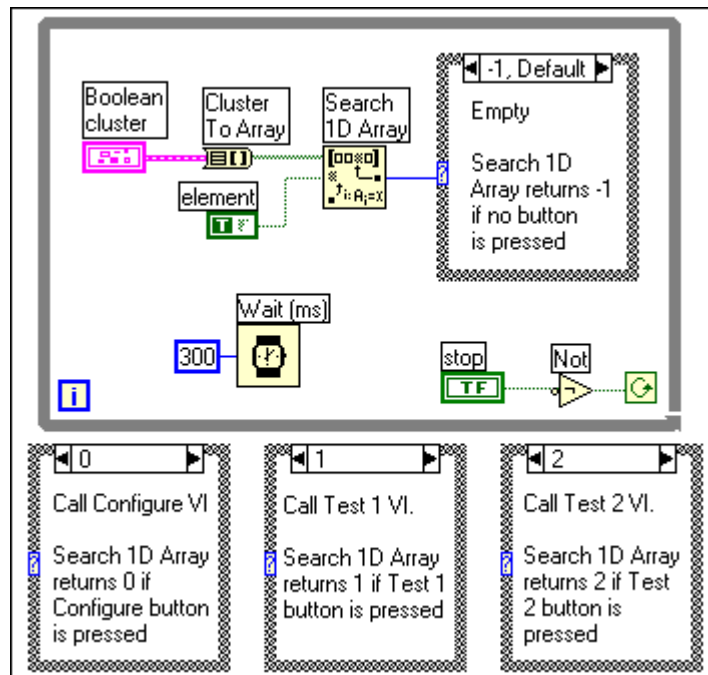
You must pop up on the function icon to set the number of components in the cluster. The default number of components in the output cluster is nine.

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 shown below.



The VI block diagram is shown below. 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 (**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 on a button, **Search 1D Array** returns an index value of -1. If no buttons are pressed, Case 1 is executed, which does nothing. Clicking on the Configure button executes Case 0, which could, for example, call the “Configure” subVI. Clicking on the Test 1 button executes Case 1, which could call the “Test 1” subVI, and clicking on the Test 2 button executes Case 2. The While Loop repeatedly checks the state of the Boolean cluster control until you click on the Quit button.

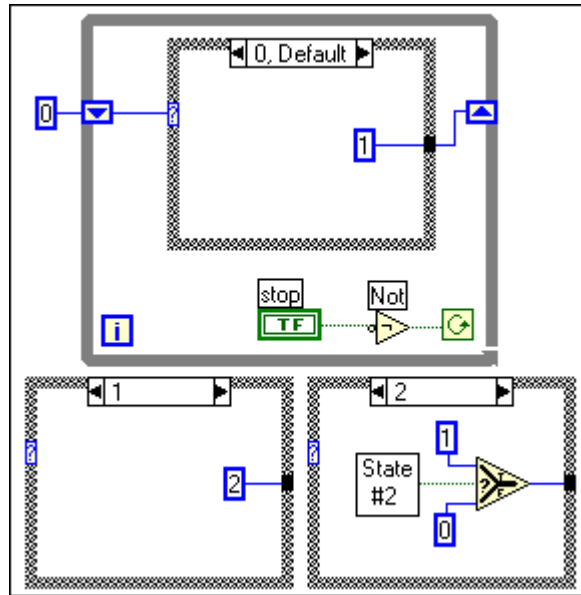


State Machine Programming

A state machine in LabVIEW 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 can 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 the VIs and other code that the state should execute within the appropriate case. A shift register stores the state to be

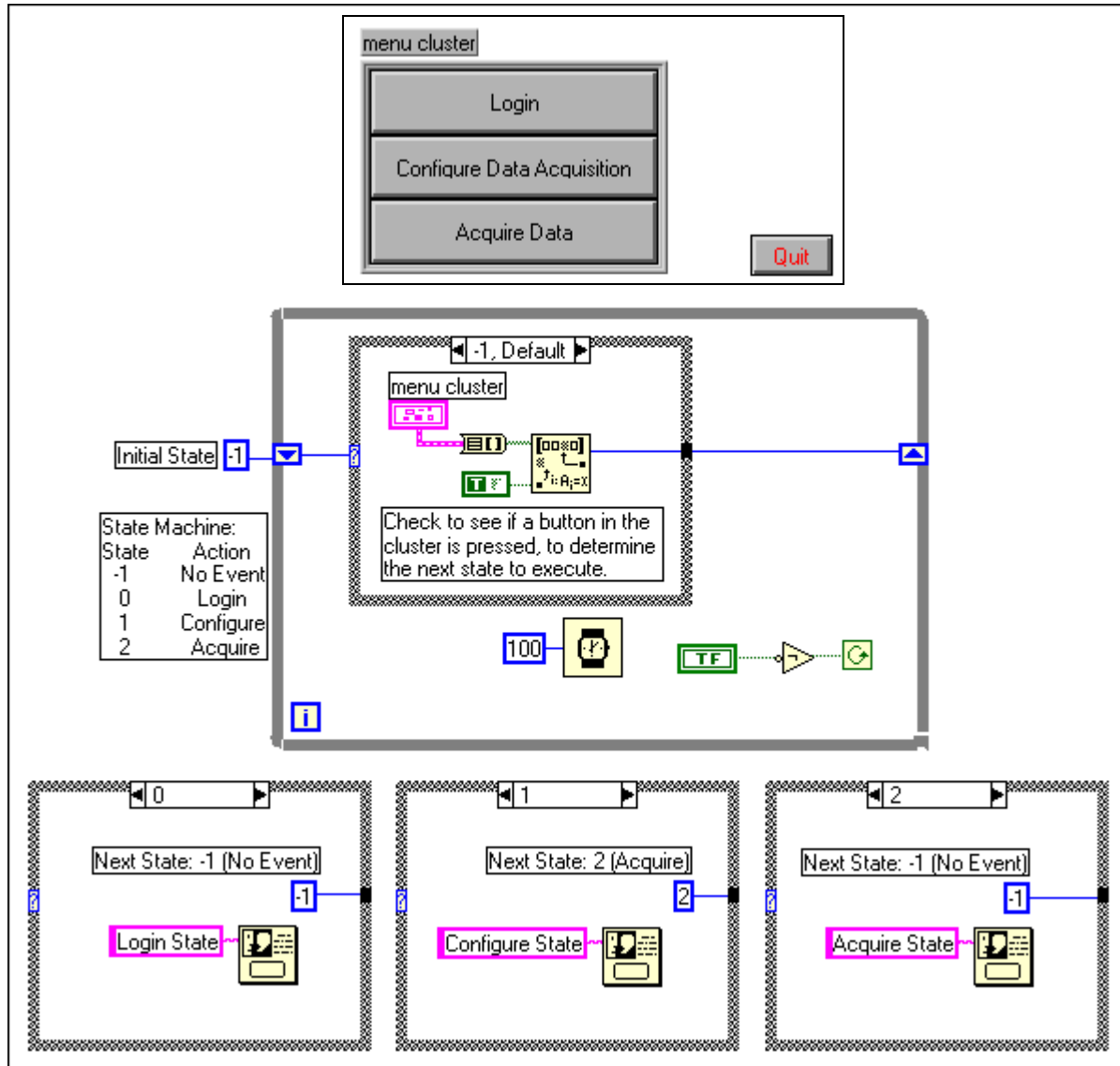
executed upon the next iteration of the loop. The diagram of a simple state machine appears below.



You can combine the concept of a state machine with a Boolean menu cluster to provide a powerful menuing system. 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)

An example state machine for this application is shown below:



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 exists 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. However, whenever the “Acquire” state is executed, the “Analyze” state should immediately follow. This is based on the flowchart developed for the application in Lesson 1.

The dependencies of the application can be described as a simple state machine, where each numeric state leads to another subsequent state. This series of dependencies can be summarized in the table below:

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	Configure	Configure acquisition	No Event
2	Acquire	Acquire data	Analyze (3)
3	Analyze	Analyze data, possibly save to file	No Event
4	View	View saved data files	No Event

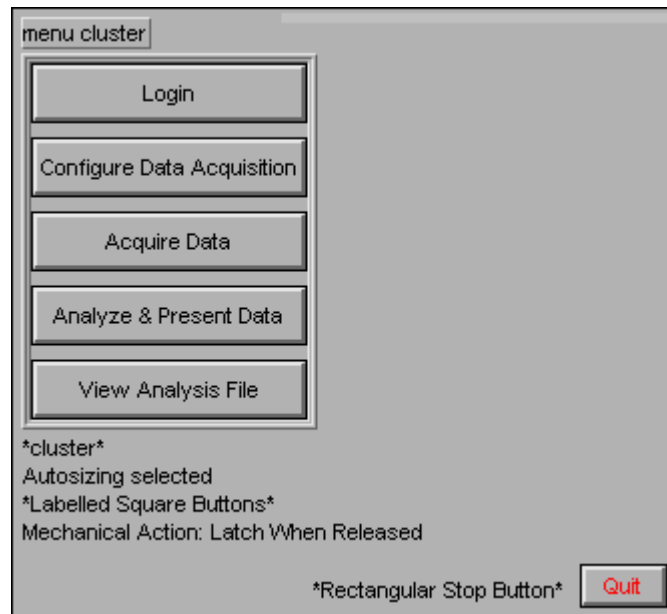
In this exercise, you will finish building the state machine to be used in this application and observe its operation.



Note

You will use this VI in the project in Lesson 6.

Front Panel

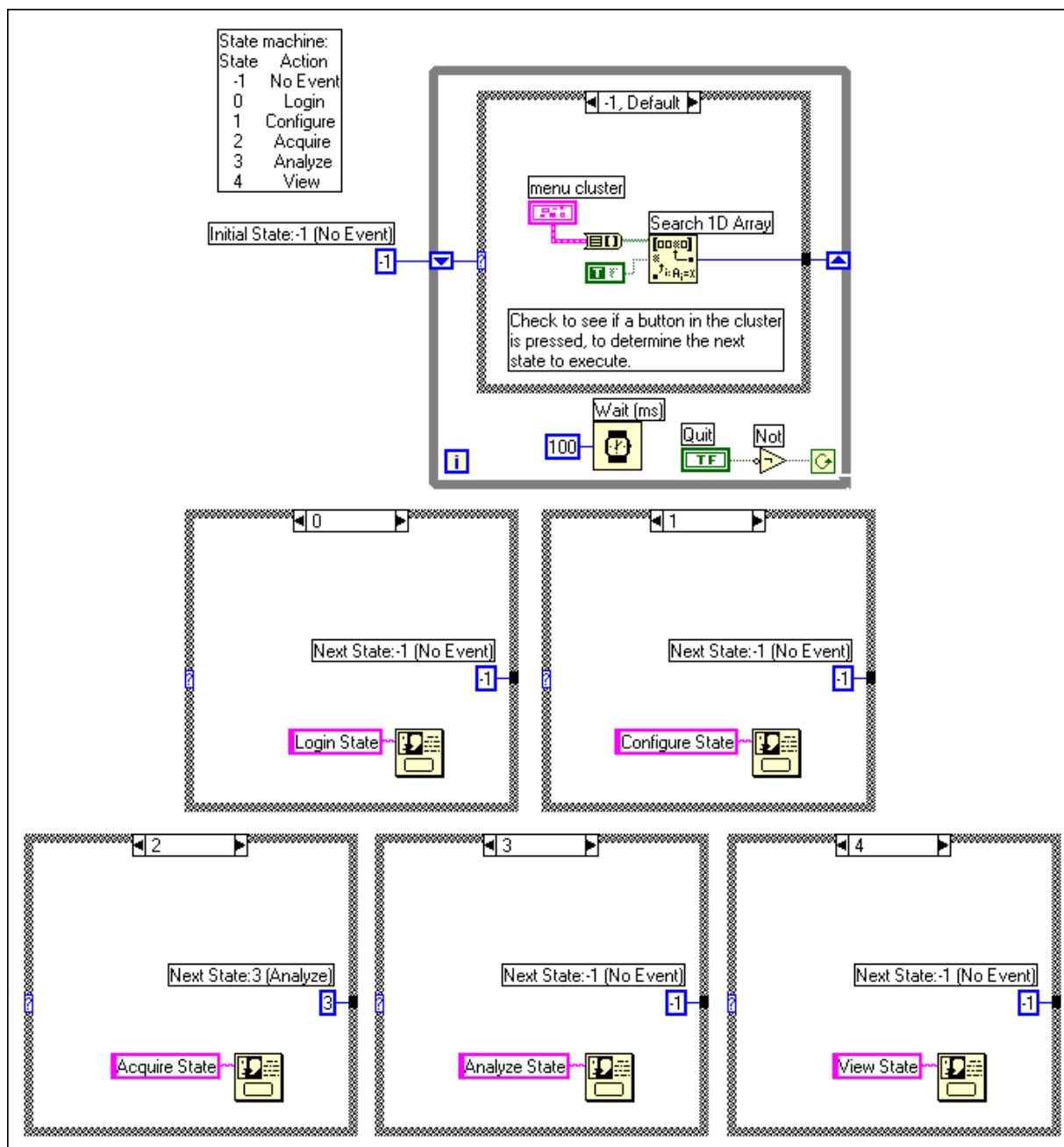


1. Open the **Menu** VI located in `Basics2.11b`. The front panel and block diagram are partially complete.
2. Complete the front panel according to the above figure. Each button in the cluster will trigger an appropriate state whenever it is pressed. When building the front panel, make sure that the Login button is at cluster order 0, Configure Data Acquisition is cluster order 1, Acquire Data is cluster order 2, Analyze & Present Data is cluster order 3, and View Analysis File 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. Complete the block diagram as shown above.

 **Cluster To Array** function (**Cluster** or **Array** palette). In this exercise, this function converts the cluster of Boolean buttons into an array of Booleans. The Boolean at cluster order 0 becomes the Boolean element at array index 0, cluster order 1 becomes array index 1, and so on.



Search 1D Array function (**Array** palette). 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 on a button.



One Button Dialog function (**Time & Dialog** palette). These functions are used to indicate which state has been selected and loaded into the shift register.

2. Save the VI.
3. Run the VI. Whenever you press the Login, Configure, Analyze, or View buttons, a dialog box should pop up to indicate that you are in the associated state. Whenever you press the Acquire button, however, two dialog boxes should pop-up: the first dialog box should indicate that you are in the Acquire state, and the second dialog should indicate that you are in the Analyze state.
4. Using the Single-Step and Execution Highlighting features, observe how the VI executes. Notice that until you press 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 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, as well as with the flowchart designed in Lesson 1.
In later exercises, you will substitute VIs that you create for the One Button Dialog functions to build the application.
5. Press **Quit** on the VI's front panel to halt execution.

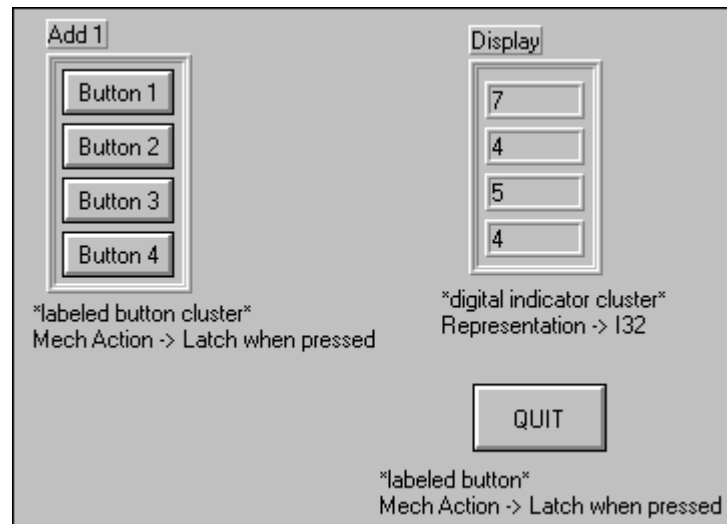
End of Exercise 2-3

Exercise 2-4 (Optional) Cluster Conversion Example.vi

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

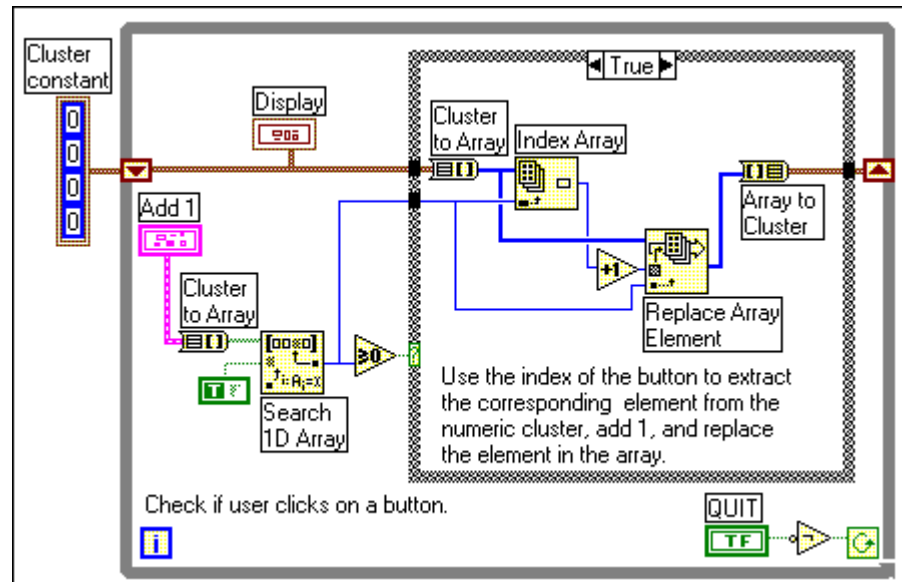
You will examine a VI that uses clusters to process data. The VI features a cluster containing four labeled buttons. The VI keeps track of the number of times you click on each button.

Front Panel



1. Open the **Cluster Conversion Example** VI in Basics2.llb.

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.

 **Cluster to Array** function (**Cluster** or **Array** palette). In this exercise, this function converts the Add 1 cluster containing four Booleans to a Boolean array with four elements. That is, each cluster button represents one element in the array.

 **Search 1D Array** function (**Array** palette). 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 on a button.

 **Greater or Equal to 0?** function (**Comparison** palette). In this exercise, this function compares the index returned by **Search 1D Array** to 0. If the index is less than 0, no button in the Add 1 cluster was pressed. An index of 0 or greater indicates which button you pressed.

 **Index Array** function (**Array** palette). In this exercise, this function indexes the element corresponding to the button that you clicked.

 **Increment** function (**Numeric** palette). In this exercise, this function increments the number the **Index Array** function returns by one.

 **Replace Array Element** function (**Array** palette). In this exercise, this function replaces the incremented number in the numeric array.



Array to Cluster function (**Cluster** or **Array** palette). In this exercise, this function converts the array to a cluster so that you can display it.

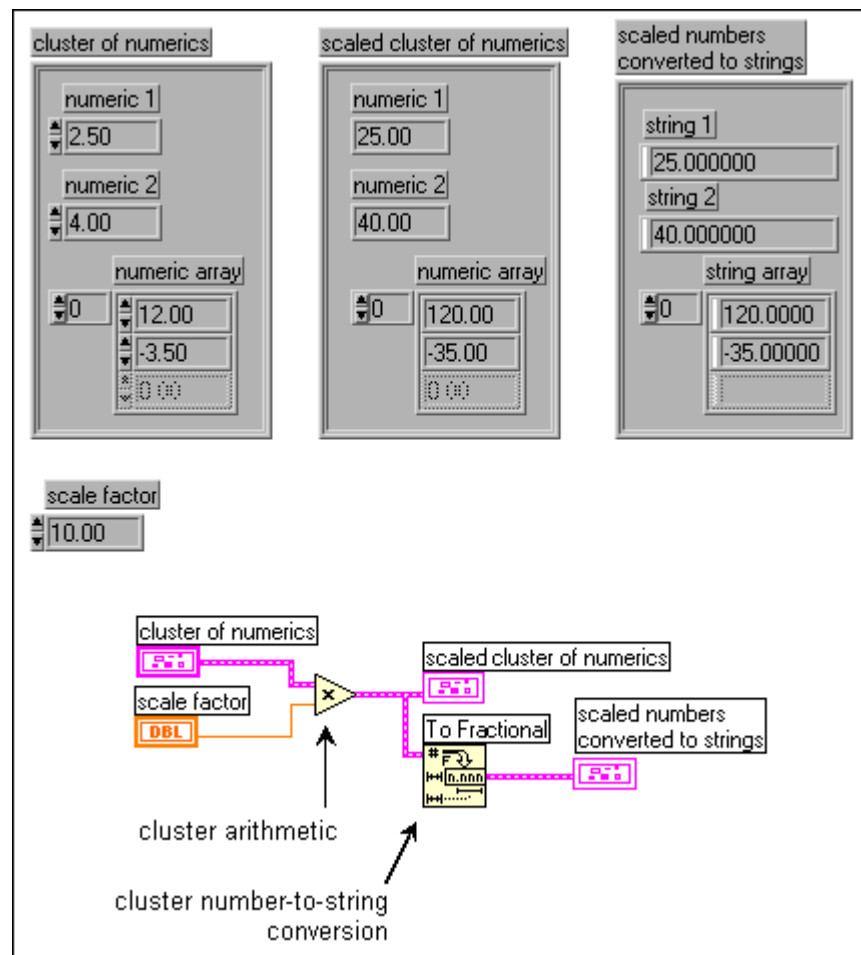
2. Run the VI. Click on a button. The corresponding digital indicator should increment each time you click on a button.
3. Close the VI. Do not save any changes.

End of Exercise 2-4

E. Using Polymorphism with Clusters

Most LabVIEW functions are polymorphic. As discussed in the LabVIEW Basics I course, polymorphic functions can have inputs of different types. You can use the **Online Reference (Help menu)** to search for a list of polymorphic functions.

Because the arithmetic functions are polymorphic, you can use them to perform computations on clusters of numerics. The string-to-number functions are also polymorphic. As shown in the following example, you use the arithmetic functions with clusters in the same way you use them with arrays of numerics. You can also use the string-to-number functions to convert a cluster of numerics to a cluster of strings.

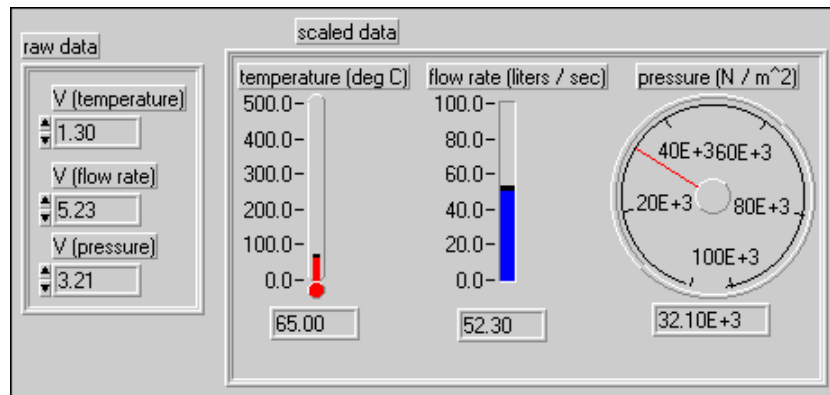


Exercise 2-5 (Optional) Cluster Scaling.vi

Objective: To build a VI that uses polymorphism with clusters.

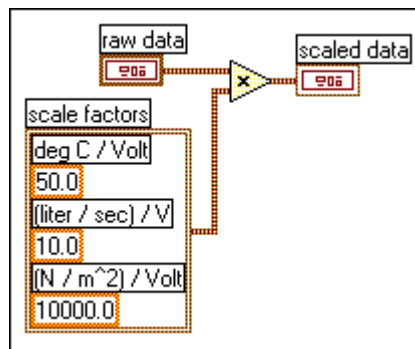
This VI will scale values stored in a cluster, where each component in the cluster has a different scale factor. In this exercise, assume that the voltages were measured from transducers that measure the pressure, flow rate, and temperature. The VI will then scale these values to get the “actual” values present in the system.

Front Panel



1. Open the **Cluster Scaling** VI in Basics2.11b. The front panel is already built for you. You will finish building the block diagram.

Block Diagram



1. Build the block diagram shown above. Make sure you apply the correct scale factors to each component in the raw data cluster.



Multiply function (Numeric palette). In this exercise, this function multiplies all the components in the cluster by different scale factors.

2. Save and run the VI. Test several alternatives to ensure that the VI works properly.
3. Close the VI.

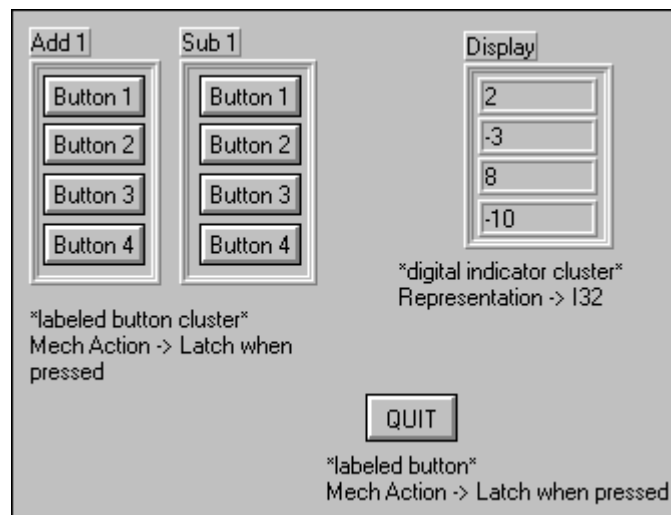
End of Exercise 2-5

Summary, Tips, and Tricks

- A cluster is a data structure that groups data, even data of different types.
- If a VI has many front panel controls and indicators that you need to associate with terminals, you may want to group them into one or more clusters and use fewer terminals.
- Objects inside a cluster must be all controls or all indicators.
- Creating a cluster on the front panel or in the diagram is a two-step process. First, create a *cluster shell* and then place the components inside the cluster shell.
- When working with clusters, the order of the components is critical. This is especially true if you have similar objects inside a cluster.
- The **Unbundle** function (**Cluster** palette) splits a cluster into each of its individual components.
- The **Bundle** function (**Cluster** palette) assembles individual components into a single cluster.
- The **Unbundle by Name** function (**Cluster** palette) extracts components from a cluster by name.
- The **Bundle by Name** function replaces components in a cluster by name.
- Error clusters are a powerful method of error handling used with nearly all of the I/O VIs 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.
- 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 (**Cluster** or **Array** palette) converts a cluster to a 1D array.
- The **Array to Cluster** function (**Cluster** or **Array** palette) converts a 1D array to a cluster. You must specify the number of components in the output cluster.
- State machine programming is very useful for user interface VIs and generates clean, simple code.
- You can use the polymorphic capabilities of LabVIEW functions with clusters. The **Online Reference** (**Help** menu) contains a list of all polymorphic functions.
- Try to place items in a cluster that logically or conceptually belong together.
- Avoid extremely complex clusters. Performance will suffer.

F. Additional Exercise

- 2-6 Modify Exercise 2-4 by adding a cluster of four labeled buttons. Each time you click on a button in the new cluster, decrement the display by one. Use the front panel shown below to get started. Save the VI as **Cluster Example 2.vi**.

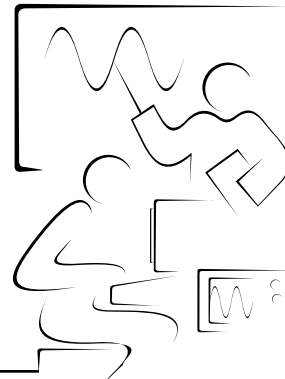


Notes

Notes

Lesson 3

Local and Global Variables



Introduction

In applications such as user interface development, you often must update or read front panel controls or indicators from more than one location in the program. For example, you may want to clear the `Login:` and `Password:` prompts every time a new user logs in. In this case, you are faced with the task of reading from these prompts when a user logs in, and then writing empty strings back to them after the user logs out. In LabVIEW, local variables lend themselves well to accomplishing such a task.

You Will Learn:

- A. How to use local variables.
- B. How to use global variables.
- C. Some tips about using local and global variables.

Application SubVIs You Will Build:

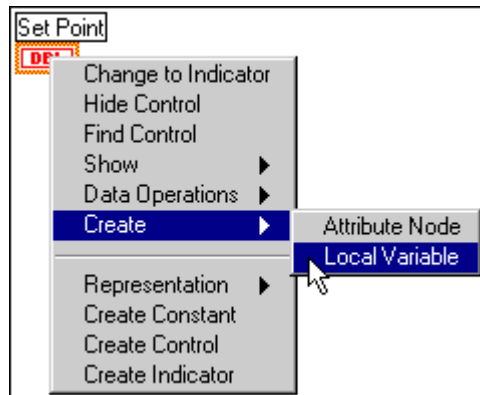
- A. `Login VI`

A. Local Variables

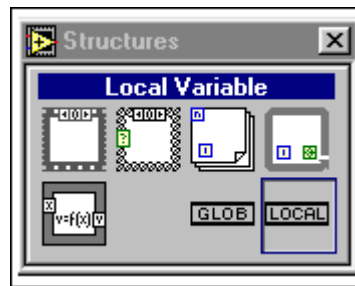
Up until now, you have read data from or updated a front panel object using its terminal on the block diagram. However, a front panel object has only one terminal on the block diagram, and you may need to update or read a front panel object from several locations on the diagram. Using local variables, 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 popping up on 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 diagram. When you pop up on a terminal to create a local variable, the local variable refers to the object you popped up on to create the icon.

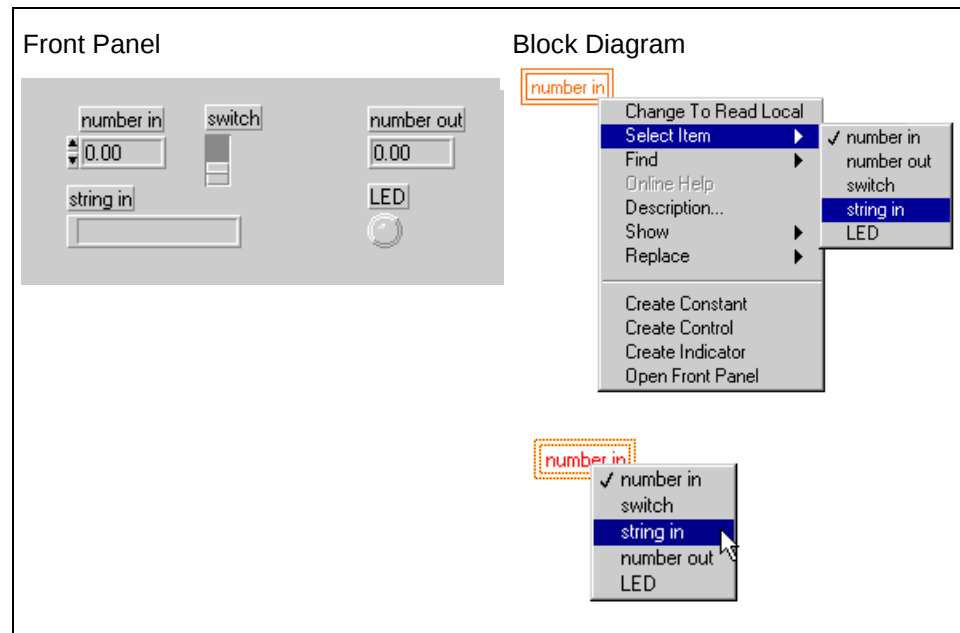


Another way to create a local variable is to select **Local Variable** from the **Structures** palette. A local variable node appears on the block diagram. The node shown at left appears if none of the controls or indicators on the front panel have an owned label. Otherwise, the local variable appears with the name of the first front panel object you created. You can select the front panel object you want to access by popping up on 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, you should always label your front panel controls and indicators with descriptive names, using owned labels.*



number in
Local variable
icon with
"number in"
selected

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 on the variable using the Operating tool, or popping up on it and choosing the front panel object from the **Select Item** menu.

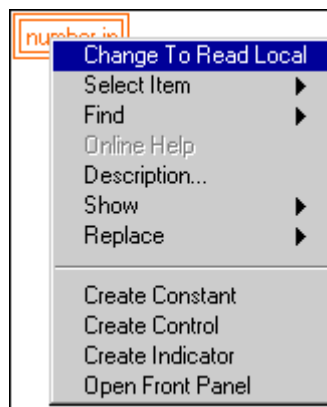


Read Locals and Write Locals

You can either read data from a local variable or write data to it. After you place the local variable on your diagram, you must decide how you will 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, pop up on the variable and select **Change To Read Local**. On the diagram, a *read local* icon behaves just like a control. When this node executes on the diagram, your program reads the data in the associated front panel control or indicator.



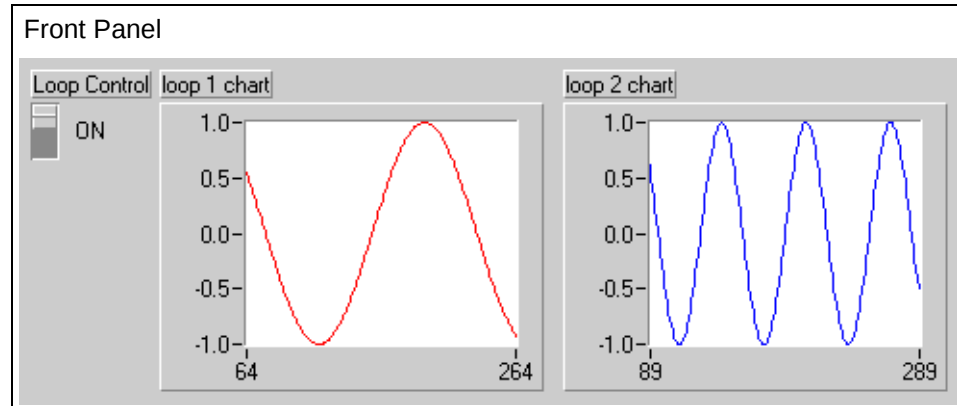
To change a *read local* to a *write local*, pop up on the variable and select **Change To Write Local**. The local variable will change its “data direction” so that it *receives* data instead of providing data.

On the 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 figure below, both local variables refer to the same item on the front panel.



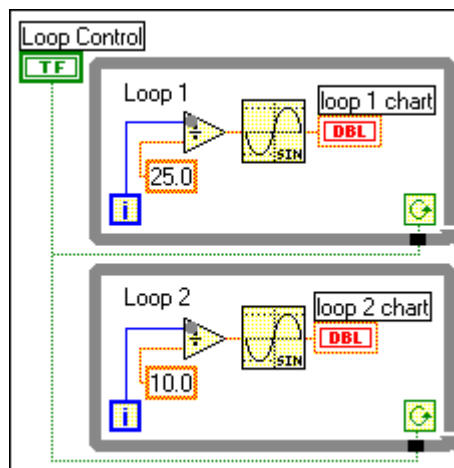
Local Variable Example

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.



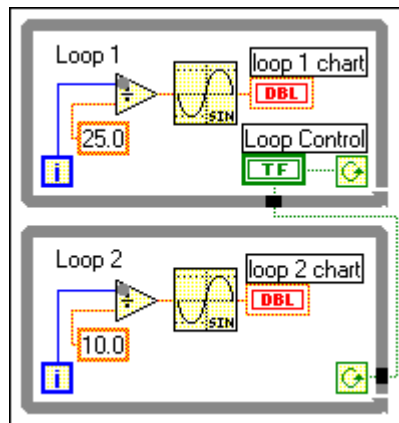
Method 1 (Incorrect)

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.



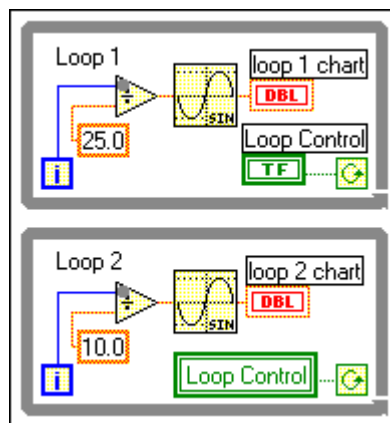
Method 2 (Incorrect)

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 also 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.



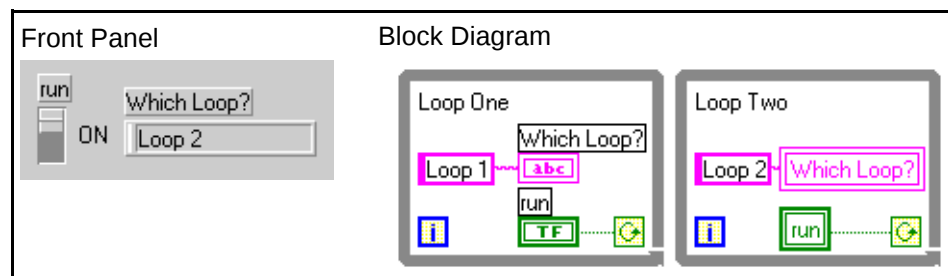
This simple example demonstrates the need for local variables. As previously shown, using the local variable gives access to a single front panel object from several locations on the block diagram. Local variables are also necessary when you cannot accomplish your goal using wires to carry the data.

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.

Using a local variable, however, you can access the same front panel indicator from more than one location on the diagram, so that a single indicator displays the loop that is executing. The Which Loop? indicator is placed in Loop One and a local variable instance of that indicator is placed in Loop Two. Although this example is very simple, it shows how an indicator can be updated from two separate locations on the diagram.



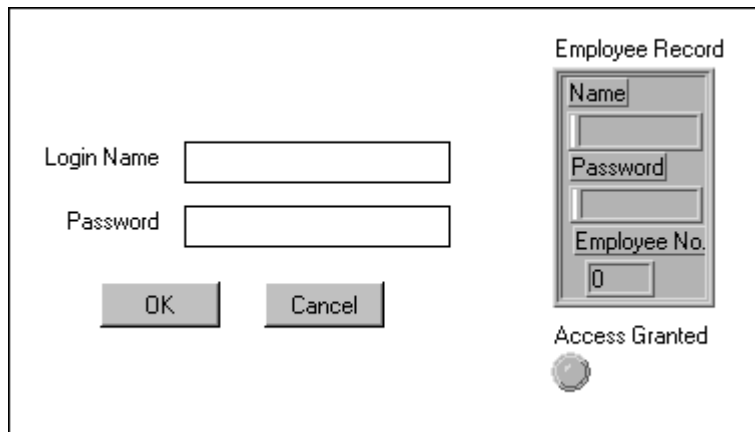
Exercise 3-1 Login.vi

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



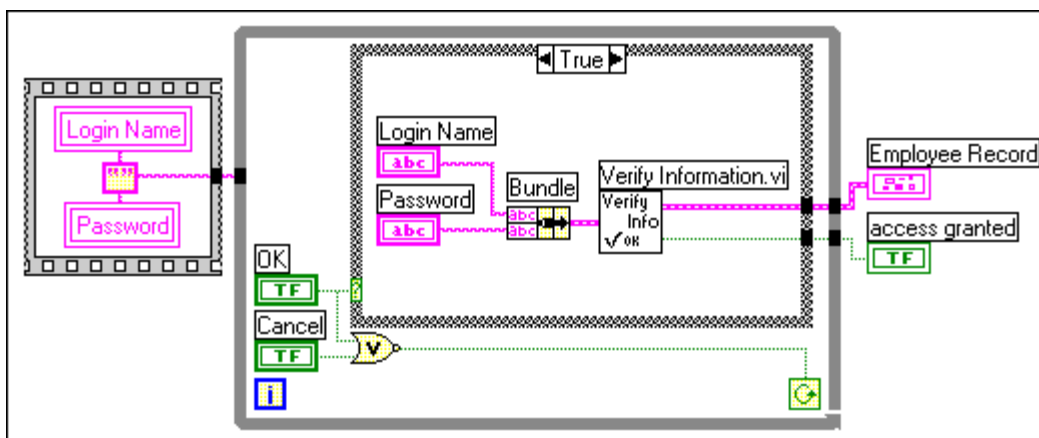
Note You will use this VI in the project in Lesson 6.

Front Panel



1. Open the **Login** VI in Basics2.11b. The front panel of the VI is already created. You will finish building the block diagram.

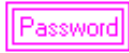
Block Diagram



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 local variable set to write local. Used to reset the login name to an empty string. Pop up on the Login Name terminal and choose **Create»Local Variable** from the pop-up menu to create this local variable.



Password local variable set to write local. Used to reset the password string to an empty string. Pop up on the Password terminal and choose **Create»Local Variable** from the pop-up menu to create this local variable.



Empty String constant (**String** palette). Used to pass string values to the Login Name and Password local variables.

**Note**

*If you have trouble wiring the string constant to a local variable, pop up on the local and select **Change to Write Local**.*

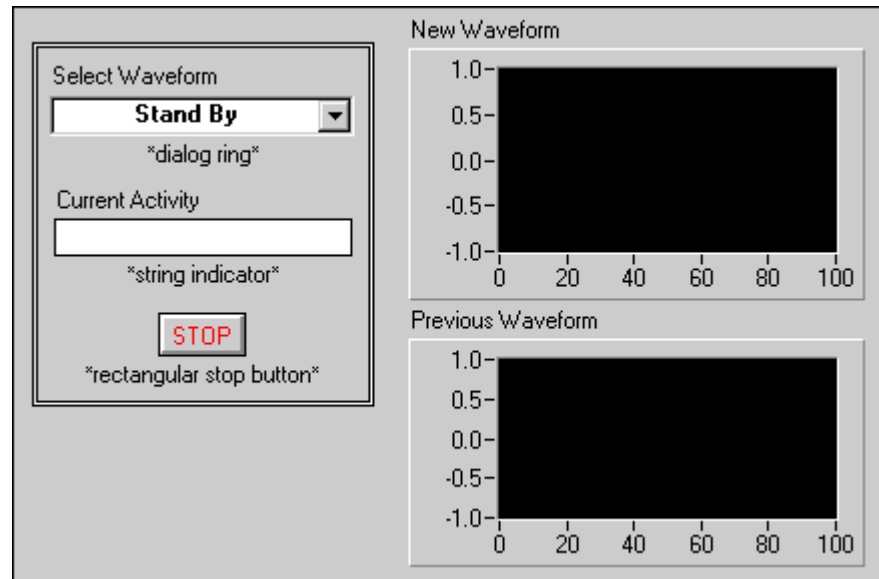
2. Return to the front panel and run the VI. Notice that the Login Name and Password controls reset to empty strings when you start the VI.
3. Save and close the VI.

End of Exercise 3-1

Exercise 3-2 (Optional) Select and Plot Waveform.vi

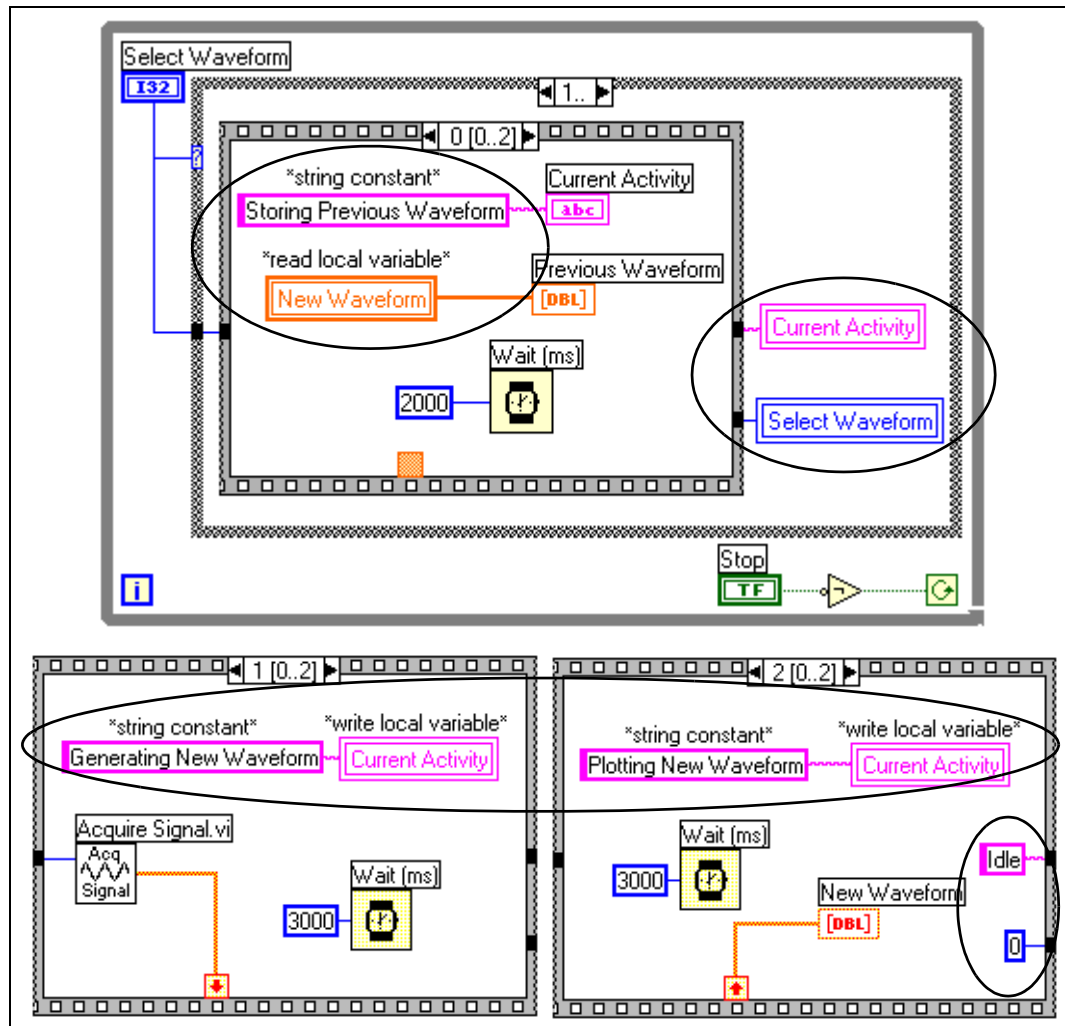
Objective: To use local variables to modify both indicators and controls on the front panel.

Front Panel



1. Open the **Select and Plot Waveform** VI in Basics2.11b. The front panel is already built. You will 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

String Constant (String palette). Used to send text values to the Current Activity front panel indicator.

123

Numeric Constant (Numeric palette). Used to reset the Select Waveform dialog ring to zero (Stand By) after the waveform has been acquired.

New Waveform

New Waveform local variable set to read local. This local variable reads the data in the New Waveform graph and passes that data to the Previous Waveform graph. Pop up on the New Waveform terminal and choose **Create»Local Variable** from the pop-up menu to create this local variable. Then, pop up on the local variable and select **Change to Read Local**.



Current Activity local variable set to write local. This local variable places new text into the Current Activity indicator. There are several instances of this local variable in the diagram, illustrating how you can access a front panel object from several locations on the diagram. To create the local variable, pop up on the Current Activity terminal and choose **Create»Local Variable** from the pop-up menu. Then, make two more copies of this local variable.



Select Waveform local variable set to write local. This local variable resets the Select Waveform control to Stand By mode. Pop up on the Select Waveform terminal and choose **Create»Local Variable** from the pop-up menu to create this local variable.



Note

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



Wait (ms) function (**Time & Dialog** palette). Generates a delay so you can observe the front panel activities.



Acquire Signal VI (**User Libraries»Basics 2 Course** palette). Generates the data for the waveform specified by the Select Waveform control.



Not function (**Boolean** palette). Inverts the value of the STOP button. Thus, the While Loop executes until you press the STOP button.

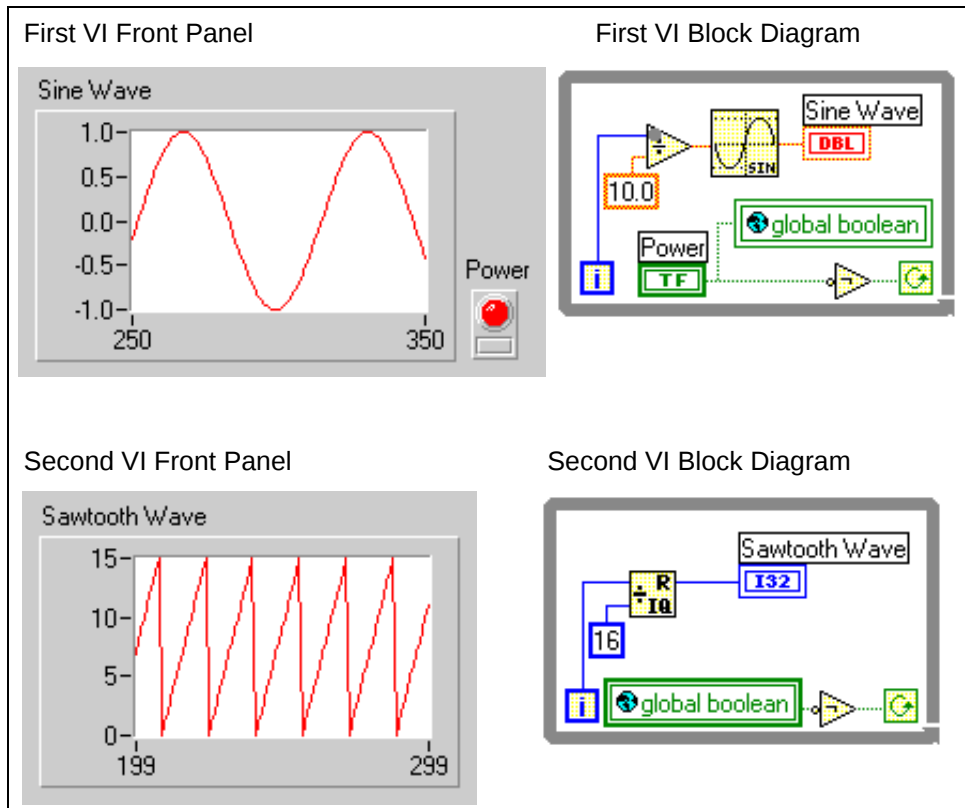
2. Return to the front panel and run the VI. Notice that the VI updates the string indicator and stores the old waveform in the Previous Waveform graph. Also, observe that after the new waveform is acquired and sent to the screen, the Select Waveform control is reset to Stand By (zero).
3. Save and close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

End of Exercise 3-2

B. Global Variables

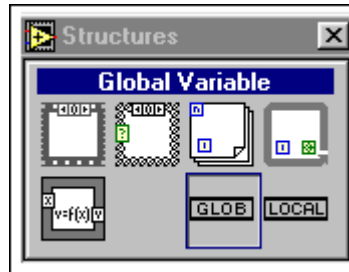
Recall that you can use local variables to access front panel objects at various locations in your block diagram. Those local variables are accessible only in that single VI. Suppose that you need to pass data between several VIs that run concurrently, or whose subVI icons cannot be connected by wires in your 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 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 diagram. The icon for a global variable on the 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.

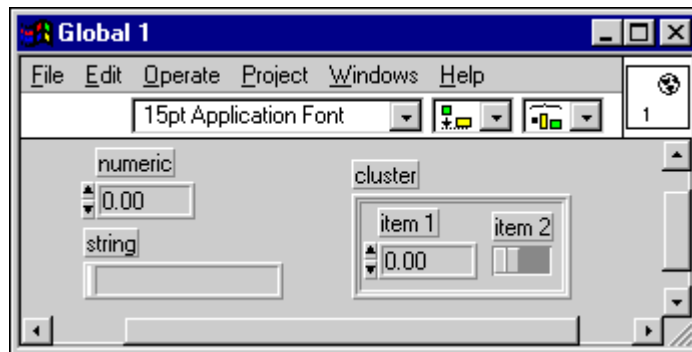
You open the panel of a global variable by double-clicking on the global variable node. You then place controls and indicators on the panel in the same way you place them on a standard VI's front panel.



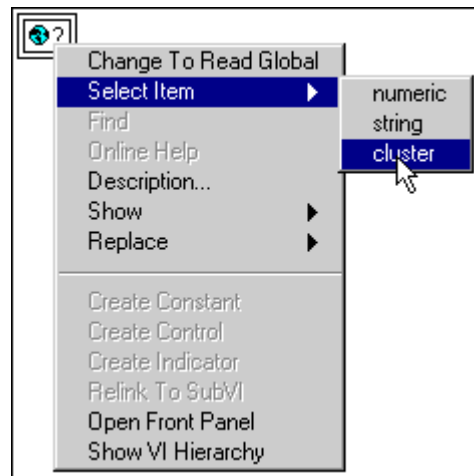
Note

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

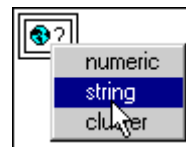
The following example shows a global variable front panel with three objects—a numeric, a string, and a cluster containing a digital and a Boolean control. Notice that the toolbar in the window does not show the same items as a normal Panel window.



After you finish placing objects on the global variable's panel, save the global variable and return to the block diagram of the original VI. You must then select the specific object in the global variable VI that you want to access. To select a specific object, pop up on 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 can also open the front panel containing the objects in the global variable from this pop-up menu.

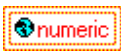


Alternately, using the Operating tool, you can simply click on the node and choose the global variable you want to access.



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 may 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, pop up on the node to select the specific object in the global variable you want to access, as described above.



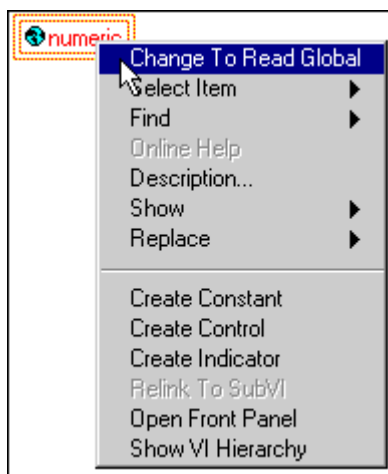
Note

A global variable 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 a global variable, or write data into it. By default, a global variable is *write global* when you create it. You can write a new value into the global variable; hence 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, pop up on the variable and select **Change To Read Global**. On the diagram, a *read global* icon behaves like a control. When this node executes on the diagram, your program reads the data in the associated front panel object.



To change a *read global* to a *write global*, pop up on the variable and select **Change To Write Global**. The global variable will change its “data direction” so that it *receives* data instead of providing data. When this node executes on the diagram, your program will send new data into the global variable.

On the 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 panel.



Exercise 3-3 Data to Global.vi

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

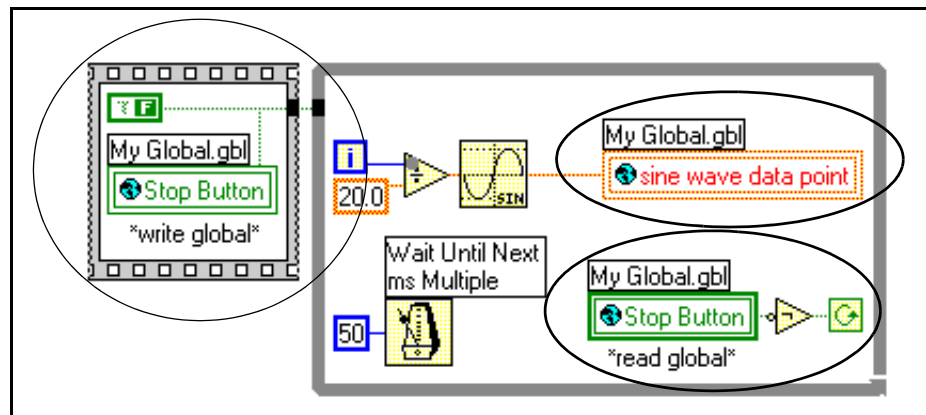
You will create a global variable in this VI and then use it to send data to the VI in the next exercise.

1. Open the **Data to Global** VI in Basics2.11b.

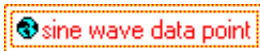
Because you will use this VI only to generate data, it has no front panel objects. You will finish building the block diagram.

2. Open the block diagram.

Block Diagram



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

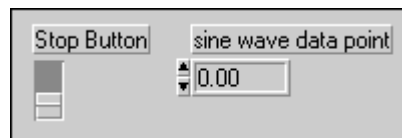


Global variable
node

My Global.gbl global variables. In this exercise, these variables pass values between two concurrently running VIs. The steps to create and configure the global variables are as follows:

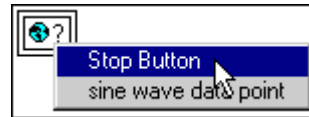
- a. In the diagram, select **Global Variable** from the **Structures** palette.
- b. Double-click on the node to bring up the global variable's panel. Create the global variable panel as shown in the following figure. *You must label the controls with the owned labels shown.* Notice that there is no block diagram associated with the global variable's front panel.

Global Variable Panel





- c. Save and close the global variable. Name it **My Global.gbl**.
- d. Return to the **Data to Global** diagram.
- e. Pop up on the global variable node and select **Show»Label**.
- f. Using the Operating tool, click on the global variable node and select **Stop Button**. Make the variable a *read* global variable by popping up on it and selecting **Change to Read Global**.



- g. Make two more copies of the **My Global.gbl** variable. Change both globals to *write globals*. Click on one variable using the Operating tool and select **sine wave data point**.



Wait Until Next ms Multiple function (**Time & Dialog** palette). In this exercise, this function ensures that data is written to the global variable every 50 ms.



Not function (**Boolean** palette). In this exercise, this function inverts the value of the Stop Button global variable so that the VI runs continuously until the global variable is TRUE.

You 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 (that is, one that has an unknown value).

2. Save the VI. Keep it open, as you will run it in the next exercise.

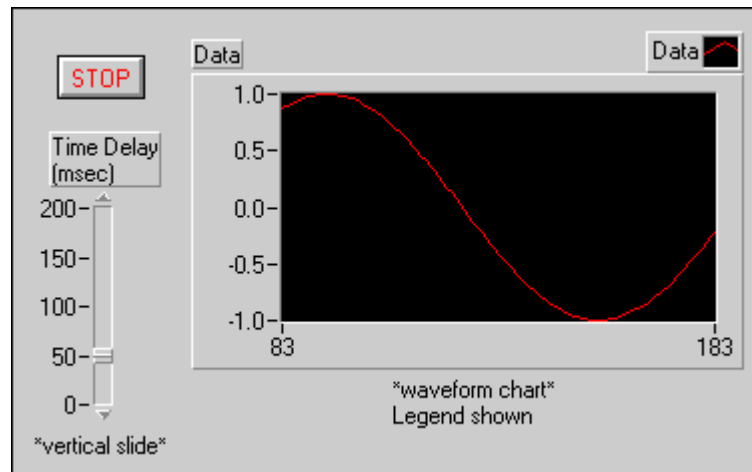
End of Exercise 3-3

Exercise 3-4 Display Global Data.vi

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

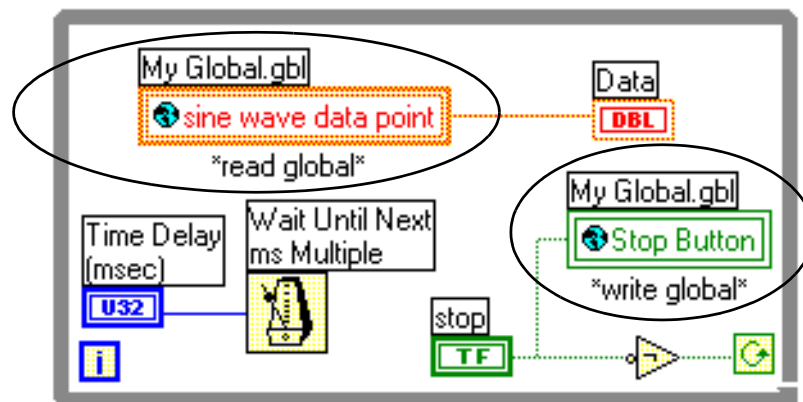
You will finish building a VI that reads data from the global variable you created in the previous exercise and displays the data on a front panel chart.

Front Panel



1. Open the **Display Global Data** VI in Basics2.11b. The front panel is already built. You will finish building the block diagram.
2. Open the block diagram.

Block Diagram



1. Finish building the portions of the block diagram shown in ellipses above.



My Global.gbl global variables. In this exercise, these variables pass values between two concurrently running VIs. These global variables

were not created from a global variable node on this block diagram, so the steps for creating and configuring them on this diagram are as follows:



- a. Choose the **Select a VI...** option from the **Functions** palette, and select **My Global.gbl** from **Basics2.11b**, in which you saved the global variable you created in Exercise 3-3. The global variable object displayed in the node will be either Stop Button or sine wave data point, depending on the order in which they were placed on the global variable's panel.
- b. Copy the global variable so that you have two instances—one Stop Button global and one sine wave data point global. To access a different global variable object, click on the node with the Operating tool and choose the desired object.
- c. Change the sine wave data point global to a *read* global by popping up on the node and choosing **Change to Read Global** from the pop-up menu.



Wait Until Next ms Multiple function (**Time & Dialog** palette). In this exercise, this function dictates the loop rate (default is 20 iterations per second).



Not function (**Boolean** palette). In this exercise, this function inverts the value of the stop Boolean control; thus, the While Loop executes repeatedly until it reads a TRUE.

The VI reads a value from the sine wave data point variable 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.

2. Save the VI.
3. Switch to the **Data to Global** VI front panel.
4. 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** is continually writing the value it calculates to the sine wave data point global variable object. **Display Global Data** is reading that 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 the Time Delay is set to 0, the same sine wave data point value is read from the global variable several times, appearing to decrease the frequency of the sine wave generated in **Data to Global**. If the Time Delay is set 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 Time Delay to 50 ms (the same rate used in the While Loop in **Data to Global**), **Display Global Data** reads and displays each value of the sine wave data point global once and 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, you must 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.

5. 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. (Keep in mind that you are inverting the value passed to the While Loop conditional terminal.)
6. Close both VIs.

End of Exercise 3-4

C. Important Advice about Local and Global Variables

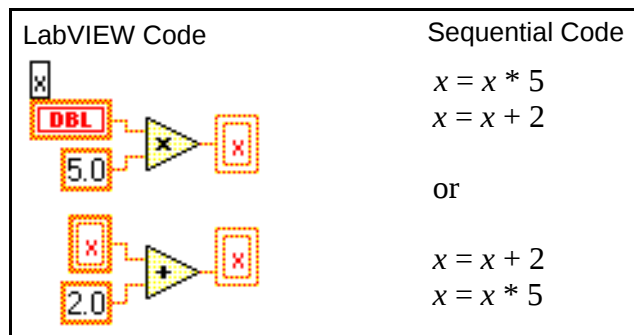
Initialize Local and Global Variables

You should verify that your local and global variables contain known data values before your program begins. Otherwise, the variables may contain data that causes your 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 conventional programming languages. Recall that LabVIEW is a *dataflow* language, not a sequential programming language. Overusing local and global variables can easily lead to unexpected behavior in your program because functions may 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 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 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 may not execute in the order you assume. If you use local and global variables, you may 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 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 may 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 some very useful goals. For example, you can create parallel loops on a single 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. Your 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, you should use local and global variables sparingly, and only when necessary.

Summary, Tips, and Tricks

- You can use global and local variables to access a given set of values throughout your LabVIEW application. These variables pass information 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 that 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.
- You should always write a value to a global variable before reading from it, so that it has a known initial value when you access it.
- You should 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; therefore, if used too frequently, they can make your diagrams more difficult for others to understand. Use locals and globals wisely.

Additional Exercises

- 3-5 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 (**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.
- 3-6 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 “sine wave data point” global faster or slower than the data was actually available.

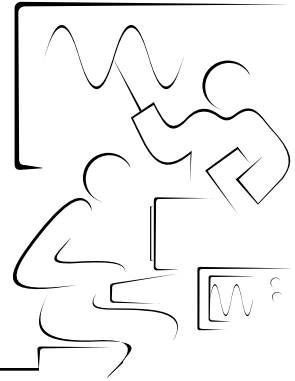
Modify **Data to Global** and **Display Global Data** (from Exercises 3-3 and 3-4) so they use a handshaking protocol to make sure that no data points are skipped or read multiple times. Set up an additional Boolean in **My Global.gbl** named **Handshake**, which will be used to indicate when VIs are ready to send or receive data. In the **Data to Global** VI, make sure the **Handshake** global Boolean is set **FALSE** before it passes a new data point to the “sine wave data point” numeric. In **Display Global Data**, check to see that the **Handshake** Boolean is set **TRUE** before it attempts to read the numeric data. In both VIs, you will need to make sure that they set the **Handshake** Boolean so the other VI knows when to access the variable.

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

Notes

Lesson 4

Attribute Nodes



Introduction

In some applications, you may 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 may 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 may 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 can also be done programmatically using attribute nodes.

You Will Learn:

- A. What attribute nodes are and how to use them.
- B. Some attributes of common front panel objects.
- C. How to use graph attributes.

Application SubVIs You Will Build:

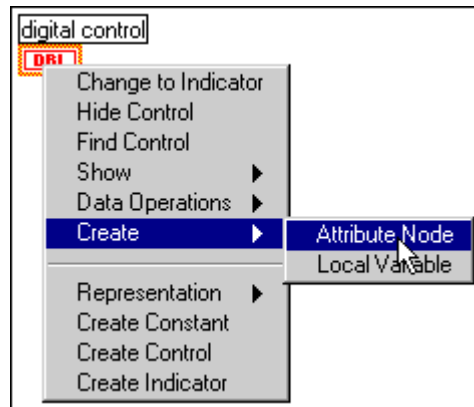
- A. **Access Verification VI**
- B. **Acquire Data VI**
- C. **Analyze and Present Data VI**

A. Attribute Nodes

Attribute nodes are special block diagram nodes that you can use to control the appearance and functional characteristics of controls and indicators. You can set attributes such as display colors, visibility, position, blinking, menu strings, graph or chart scales, graph cursors, and many more.

Creating Attribute Nodes

You create attribute nodes by selecting the **Create»Attribute Node** option from the pop-up menu of a front panel object or from its terminal on the block diagram. Selecting the **Create»Attribute Node** option creates a new node on the block diagram near the terminal for the object. If the object has an owned label, the attribute node has the same label. You can change the label after creating the node. You can create any number of attribute nodes for a front panel object.

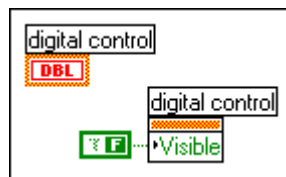


Because there are many different attributes for front panel objects, we will cover only the common attributes. Use the **Online Reference (Help menu)** to find information on a particular attribute.

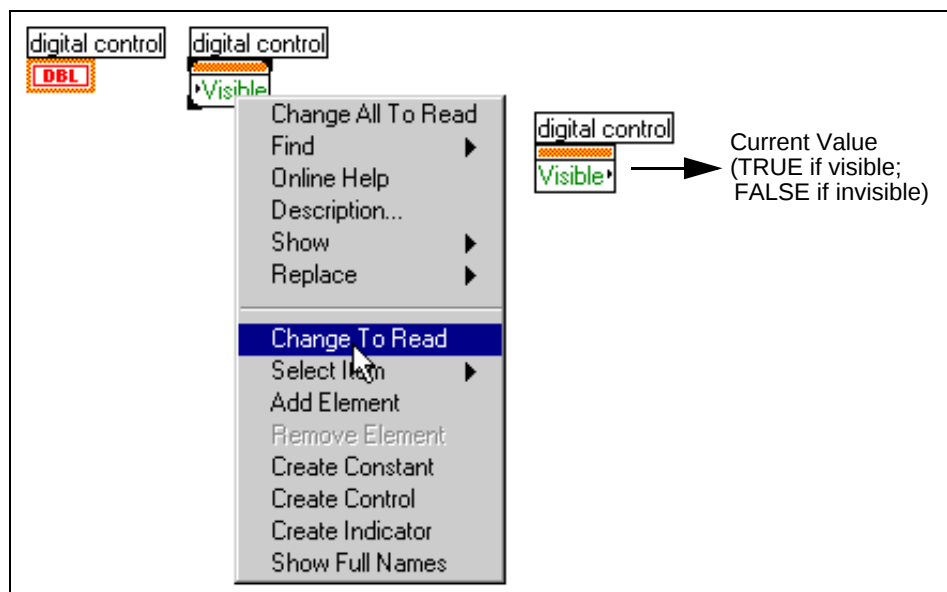
Using Attribute Nodes

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

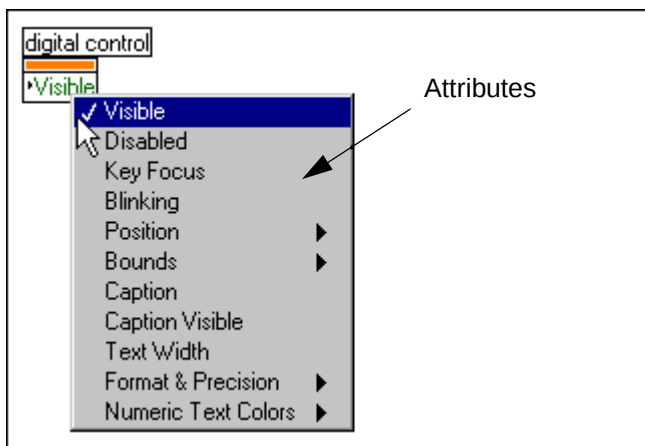
For example, if you create an attribute node for a digital control, it appears on the block diagram with the Visible attribute showing in its terminal. A small arrow appears on the left side of that terminal, indicating that you may *write* a value to the Visible attribute for the digital control. Wiring a Boolean FALSE to that terminal will cause the digital control to vanish from the front panel when the attribute node receives the data. Wiring a TRUE causes the control to reappear.



You can *read* the current state of an attribute by popping up on the attribute node terminal and selecting **Change to Read**. When the attribute node is called, it will output a Boolean TRUE if the control is visible or a Boolean FALSE if it is invisible.

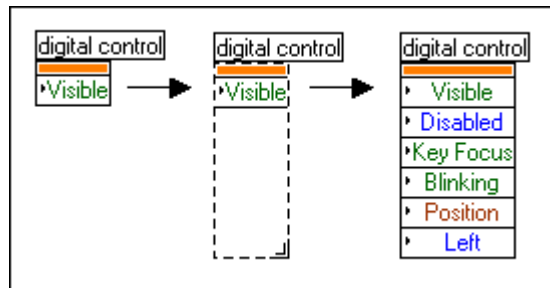


From the terminal on the attribute node, you can select attributes by clicking on the node with the Operating tool and choosing the desired attribute from the pop-up menu.

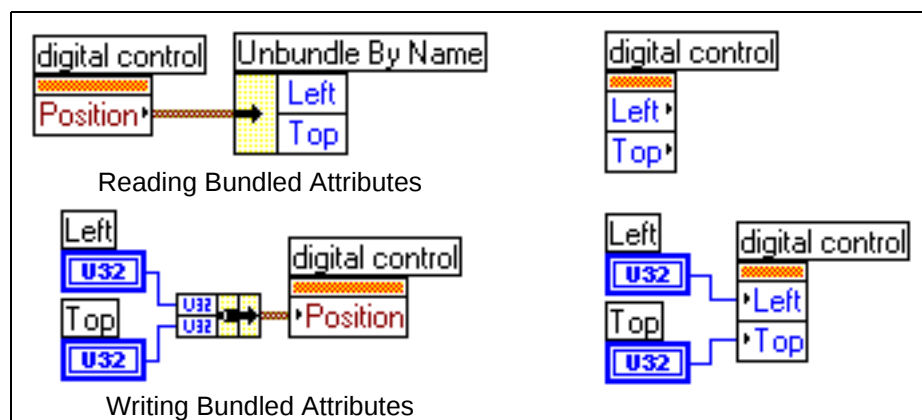




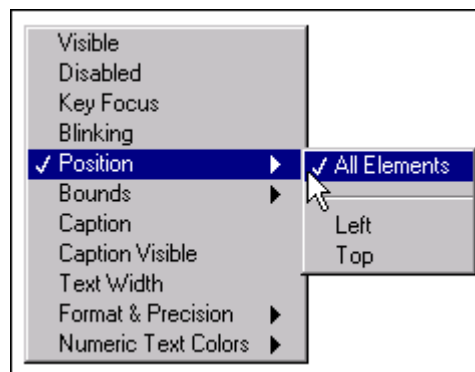
You can read or set more than one attribute with the same node by enlarging the attribute node. Using the Positioning tool, click on the lower corner of the attribute node and drag the corner down to enlarge the node. As you enlarge the node, you add more terminals. You can then associate each attribute node terminal with an attribute from its pop-up menu.



Some attributes are clusters. These clusters contain several attributes that you can unbundle using the **Unbundle** function (**Cluster** palette). Writing to these attributes requires the **Bundle** function, as shown below:

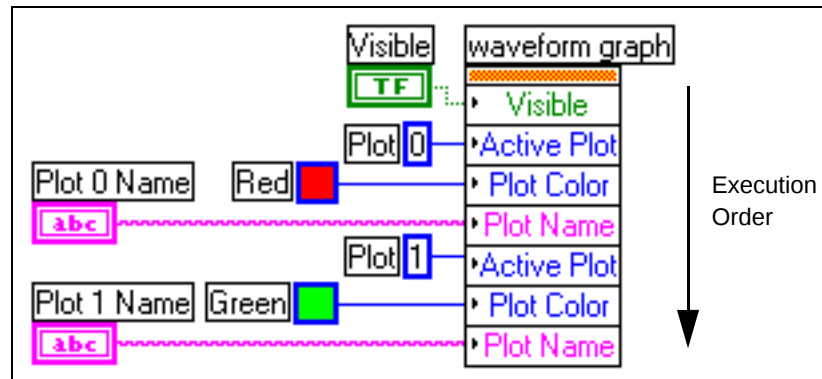


To access bundled attributes, select the **All Elements** option from the attribute's pop-up menu. For example, you can access all the elements in the Position attribute by selecting **Position»All Elements**, as shown to the right. You can also access a single element from that bundle, such as **Left**.



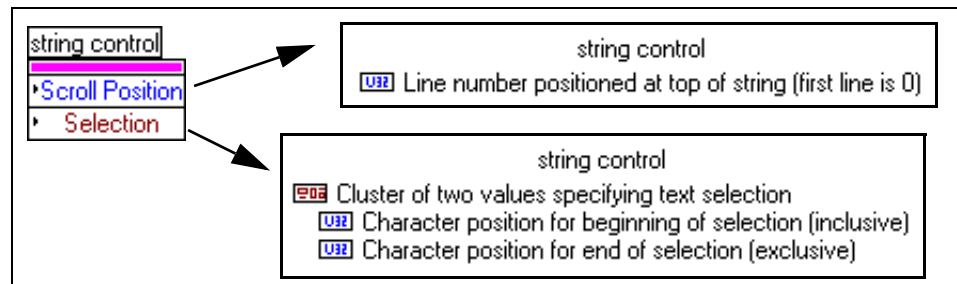
Attribute Node Execution Order

Attribute node terminals execute from the top down. For example, in the waveform graph attribute node shown below, 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 Help Window

LabVIEW's Help window and **Online Reference (Help menu)** are invaluable tools for implementing attribute nodes. You can use them to find descriptions, data types, and acceptable values for attributes. With the Help window active, pass the cursor over terminals in the attribute node to display information. The following diagram shows help for both a single attribute and a cluster of several attributes.



B. Common Attributes

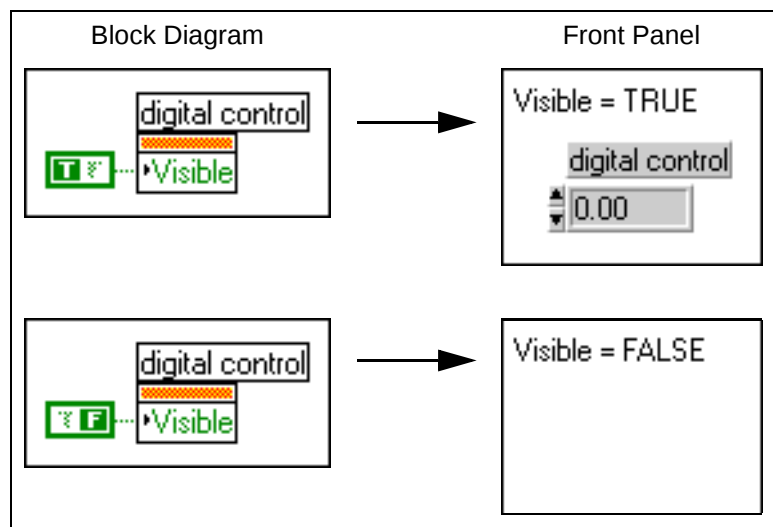
There are many attributes available for the various front panel objects in LabVIEW. This section discusses the Visible, Disable, Key Focus, Blink, Position, and Bounds attributes, which are common to nearly all front panel objects. It also introduces some attribute nodes for specific kinds of controls and indicators.

Visible Attribute



The Visible attribute 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.

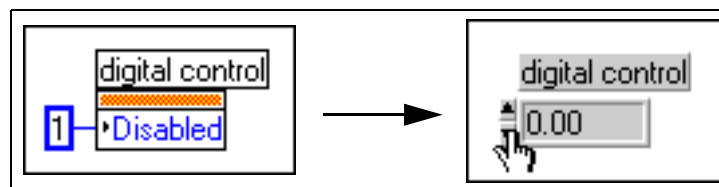


Disabled Attribute

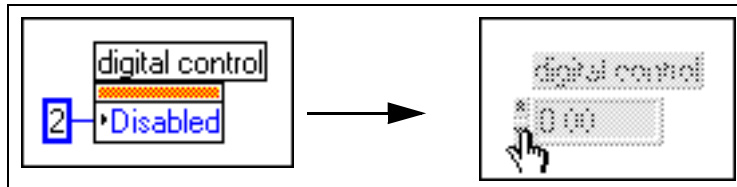


The Disabled attribute sets or reads the user access status of an object. A value of 0 enables an object so that the user can operate it. A value of 1 disables the object, preventing operation. A value of 2 disables and greys out the object.

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.

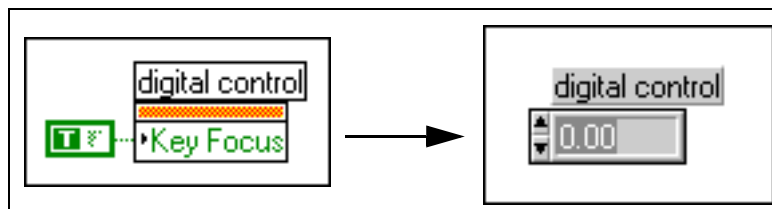


Key Focus Attribute



The Key Focus attribute 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 can also 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 mouse.

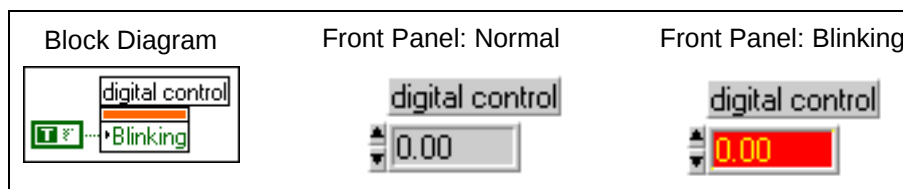


Blinking Attribute



The Blinking attribute reads or sets the blink status of an object. By setting this attribute to TRUE, an object will begin to blink. The blink rate and colors are set in the LabVIEW preferences. When this attribute is set to FALSE, the object stops blinking.

Wiring Example—Enable blinking for the digital control.

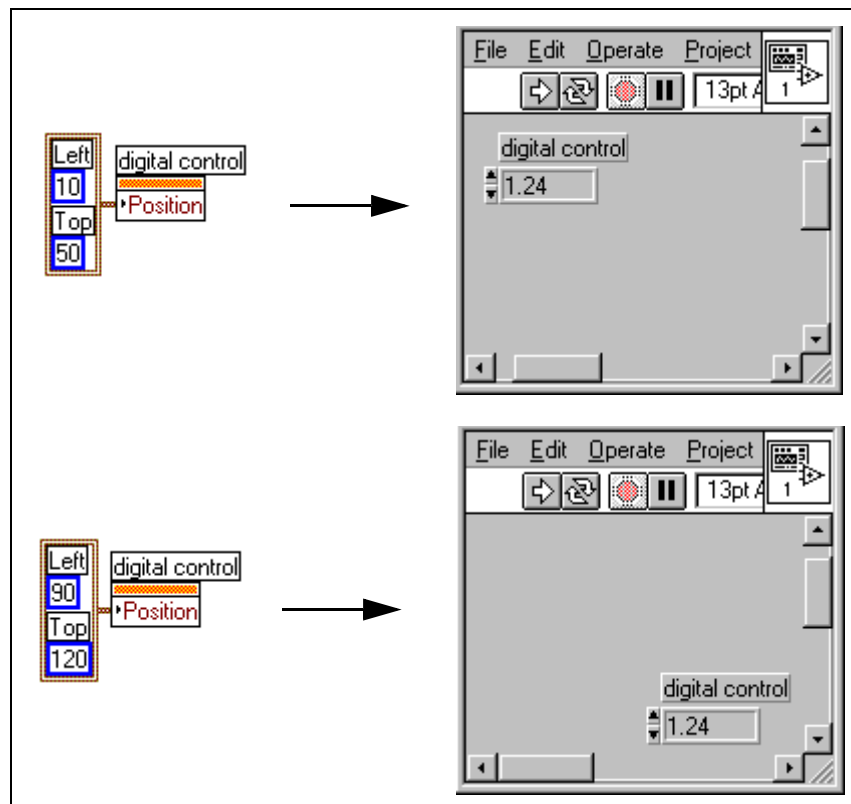


Position Attribute



The Position attribute 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 Panel window. This attribute 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 Panel window, 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 Panel window.

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

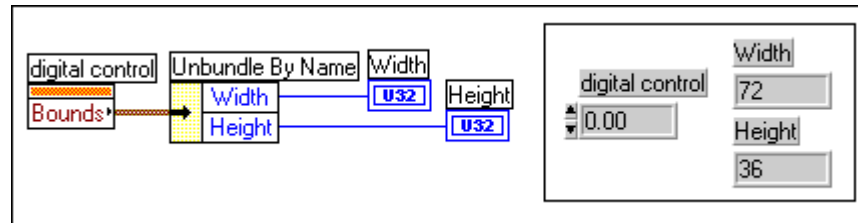


Bounds Attribute

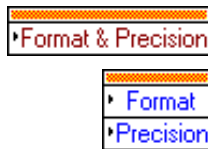


The Bounds attribute reads the boundary of an object on the front panel in units of pixels. The value includes the control and all of its parts, including the label, legend, scale, and so on. This attribute consists of a cluster of two unsigned long integers. The first item in the cluster (Width) is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

Wiring Example—Determine the bounds of the digital control.



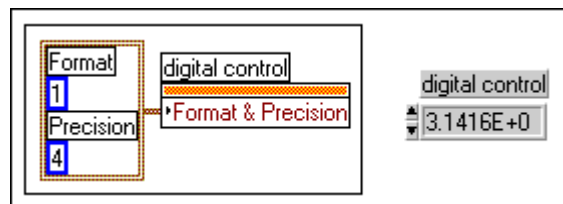
Numeric Attribute: Format and Precision



The Format and Precision attribute 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 attribute 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 Attribute: Strings [4]



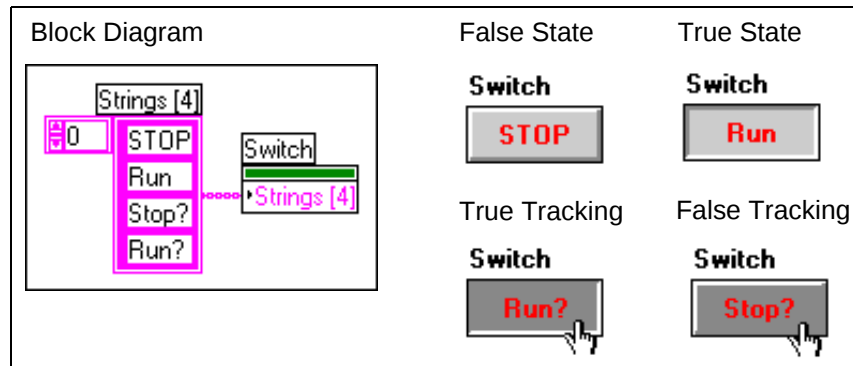
The Strings [4] attribute sets or reads the labels on a Boolean control. The input is an array of four strings that correspond to the False, True, True Tracking, and False Tracking states.

True and False: On and Off states of the Boolean.

True and False Tracking: Temporary transition levels between the Boolean states. True Tracking is the transition state when the Boolean is changed from True to False. The tracking applies only to Booleans 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?

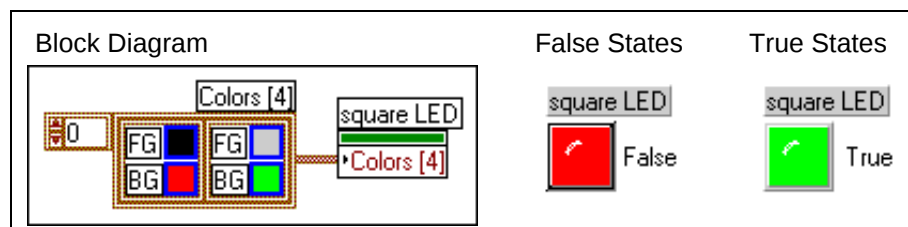


Boolean Attribute: Colors [4]



The Colors [4] attribute sets or reads the colors for foreground and background colors in each Boolean mode. The input consists of an array of four clusters where each cluster contains two integers representing color.

Wiring Example—Set the display colors for the square LED. The selection shown displays a black foreground with a red background for the FALSE states, and a grey foreground with a green background for the TRUE states.

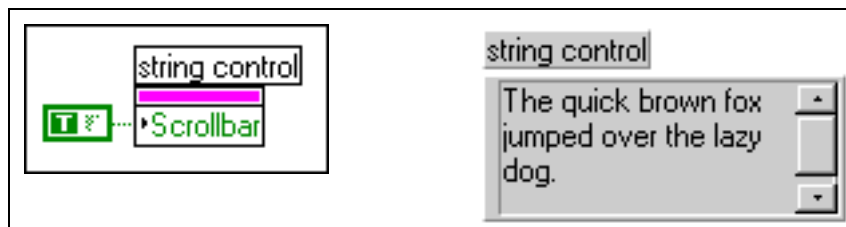


String Attribute: Scrollbar



The Scrollbar attribute sets or reads the visibility of a scrollbar for a string control or indicator. When TRUE, the scrollbar is visible.

Wiring Example—Show the scrollbar for the string control.



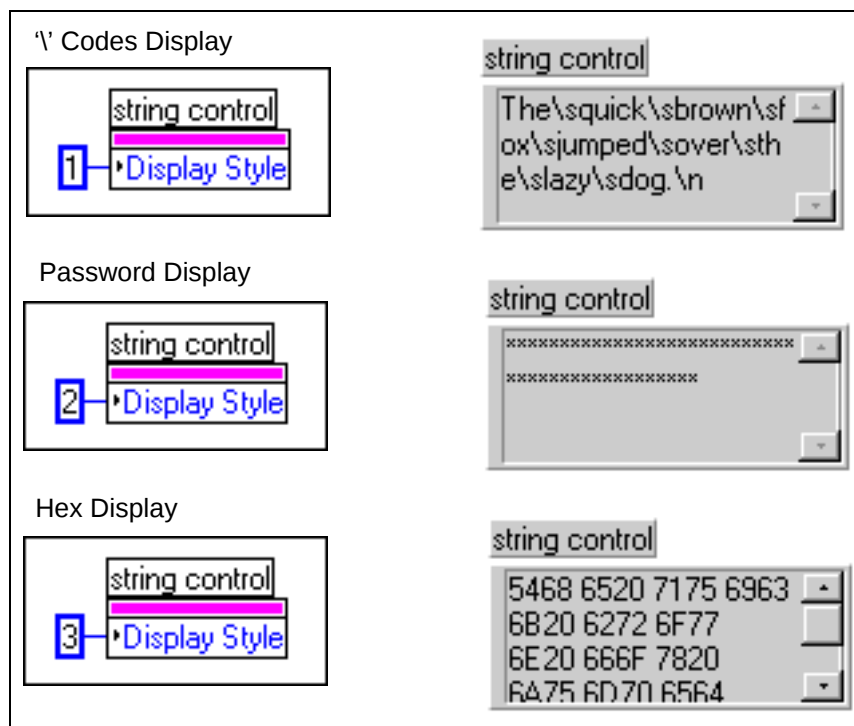
String Attribute: Display Style



The Display Style attribute 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

Wiring Example—Show the text in the string control in the three other modes.

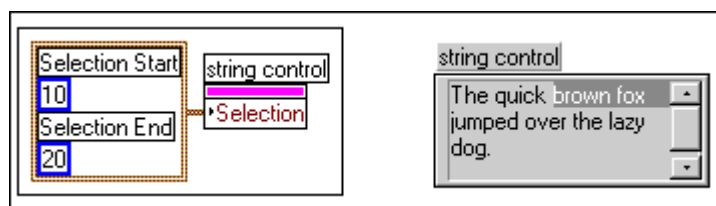


String Attribute: Selection



The Selection attribute sets or reads the selected text in the string control or indicator. This attribute is a cluster of two unsigned long integers, identifying the first and last characters selected in the string. The object must have the key focus for the selection to be highlighted.

Wiring Example—Select characters 10 through 20 in the string control.



Exercise 4-1 Access Verification.vi and Login.vi

Objective: To create and use attribute nodes.

You will finish the **Access Verification** VI, which you will use to complete the **Login** VI that you started in Exercise 3-1.

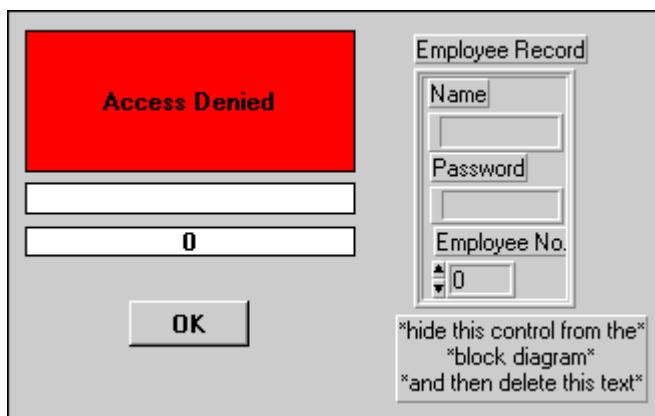


Note

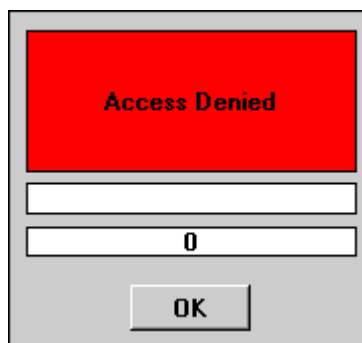
You will use this VI in the project in Lesson 6.

Part A

Front Panel

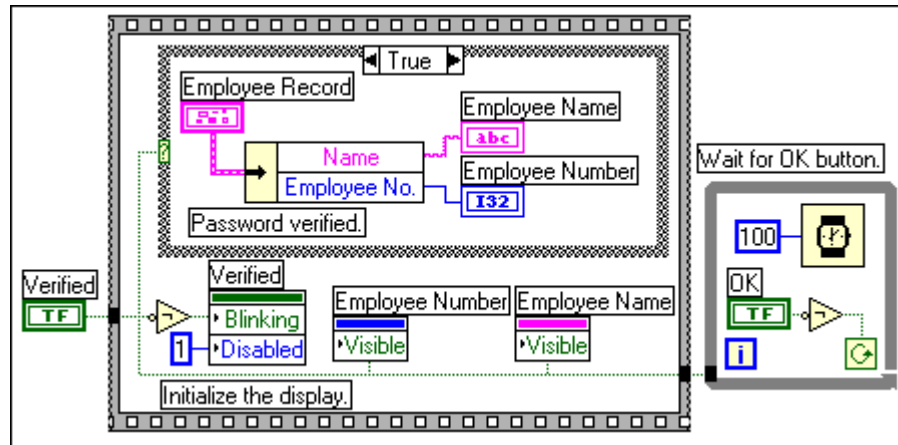


1. Open the **Access Verification** VI in Basics2.11b. The front panel of the VI is already created. You will finish building the block diagram, hide the Employee Record control, and resize the front panel.
2. Resize the window so the Employee Record control and the free label below it are not visible, as shown below.

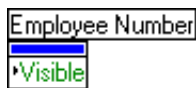


3. Open the block diagram.

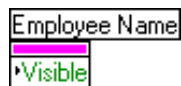
Block Diagram



1. Complete the VI diagram so that the VI behaves as follows:
 - a. The Verified control should be disabled when the VI is running.
 - b. When the Verified control is in the False state (red, Access Denied), the VI should behave as follows:
 - The Verified control should blink.
 - The Employee Name indicator should be hidden.
 - The Employee Number indicator should be hidden.
 - c. When the Verified control is in the True state (green, Access Granted), the VI should behave as follows:
 - The Employee Name indicator should be visible and show the name stored in the Employee Record cluster.
 - The Employee Number indicator should be visible and show the Employee No. stored in the Employee Record cluster.
 - The Verified Boolean should *not* blink.



To create the Visible attribute for the Employee Number indicator, pop up on the Employee Number indicator and select **Create»Attribute Node** from the pop-up menu.



To create the Visible attribute for the Employee Name indicator, pop up on the Employee Name indicator and select **Create»Attribute Node** from the pop-up menu.



To create the Blink and Disabled attributes for the Verified control, pop up on the Verified control and select **Create»Attribute Node** from the pop-up menu. Then, resize the attribute node so that it has two components. Use the Operator tool or pop up on each element in the attribute node to select the Blink and Disabled attributes.



Unbundle by Name function (**Cluster** palette). In this exercise, the function extracts the Name and Employee No. information from the Employee Record control.



Wait function (**Time & Dialog** palette). In this exercise, the function dictates that a loop iteration occurs every 100 ms.



Not function (**Boolean** menu). In the While Loop, the function inverts the value of the OK Boolean control; thus, the While Loop executes repeatedly until it reads a TRUE. In the Sequence structure, the **Not** function inverts the value of the Verified Boolean, so the Verified Boolean will blink if its value is FALSE, or not blink if its value is TRUE.

2. Test the VI. Before running the VI, use the Operating tool to set the Verified Boolean to the False state. Ensure that the Employee Name and Employee Number indicators are hidden, and that the Verified Boolean blinks and is disabled. Then, click OK to stop the VI, set the Verified Boolean to TRUE, and run the VI. The Employee Name and Employee Number indicators should be visible, and the Verified Boolean should not blink.
3. Save the VI with the changes you have made, and then close it.



Note

Be sure to close this VI after saving it. Access Verification is set up to Show Front Panel when Called and Close Afterwards if Originally Closed, because it will be used as a subVI in Part B.

Part B



Note

This VI will be used in a later exercise.

Complete the **Login** VI using the **Access Verification** VI completed in Part A.

Front Panel

1. Open the **Login** VI from Basics2.11b. You worked with this VI in Exercise 3-1. Resize the front panel so the Employee Record and Access Granted indicators are not visible.

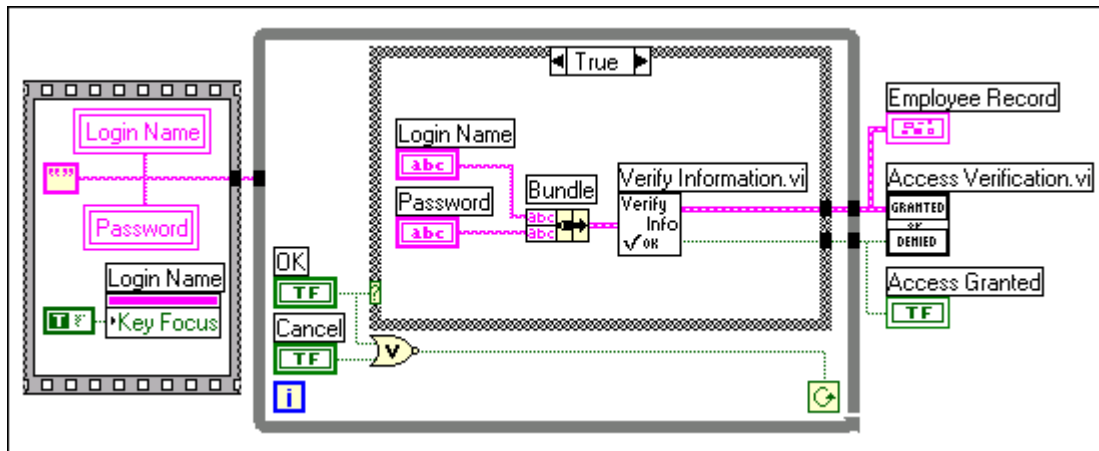


Note

If you did not complete Exercise 3-1, open Login.vi (Exercise 3.1) in Bas2Soln.llb. This library will be in the Solutions folder.

2. Open the block diagram.

Block Diagram

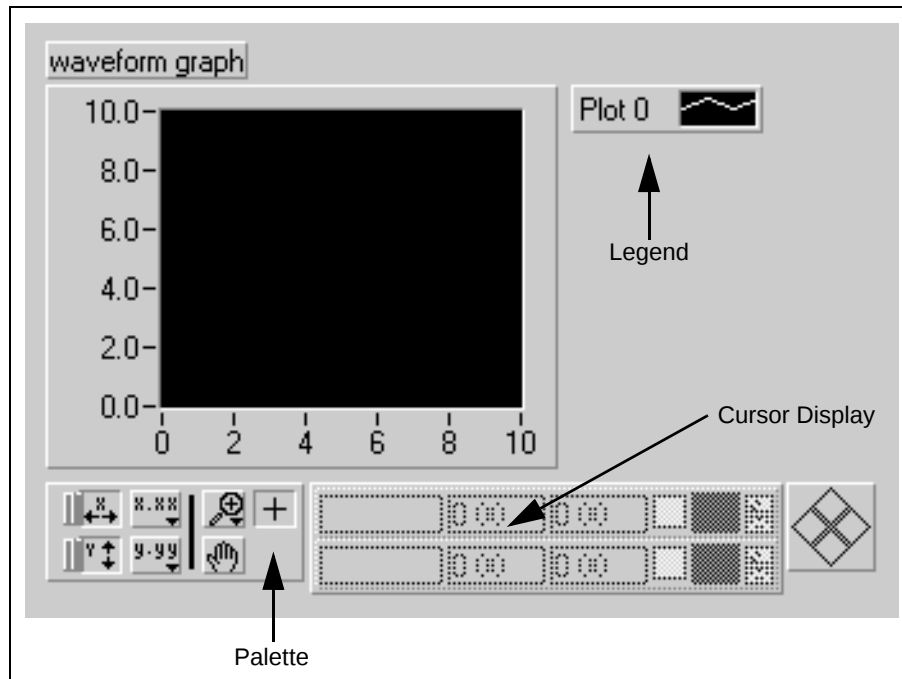


1. Add the **Access Verification VI**, which you completed in Part A, to the diagram. Use the **Select a VI...** option in the **Functions** palette. If you did not complete Part A, get the VI from the Solutions folder.
2. Set the Key Focus of the Login Name string control to TRUE before the loop starts, so when the user starts typing, the Login Name control is active.
3. Save the VI. Run it, trying several Login Name and Password combinations. Recall that valid names and passwords are stored in the **Verify Information VI**, which you created in Exercise 2-1.

End of Exercise 4-1

C. Graph Attributes

Attribute nodes greatly enhance the programmatic flexibility of graphs and charts. You can control most graph and chart features with attribute nodes. Some of these attributes are plot color, X and Y scale information, visibility of the legend and palette, size of the plotting area, and cursors.



You have many scale options for graphs and charts. With attribute 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 can also use attribute 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 attribute. 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 attribute in conjunction with other attributes that change the size of a VI's panel.

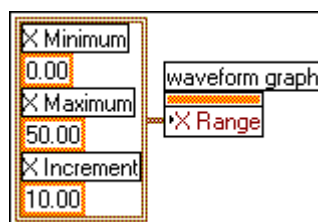
Using attribute nodes, you have access to the information that cursors provide on graphs. You use cursors to move a graphic selector on the plot. A cursor can be locked to the plot or may float on the plotting surface. Values from each cursor are returned to an optional display on the front panel. Attribute 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 Attribute



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

Wiring Example—Set the X axis range to 0 to 50 with axis increments of 10.

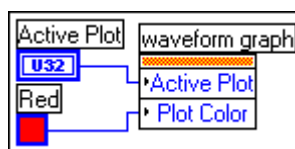


Active Plot and Plot Color Attributes



The attributes shown at left set or read the active plot (the trace for which subsequent trace-specific attributes 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 attribute is accessed by selecting **Plot Info»Plot Color** from the attribute list.

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

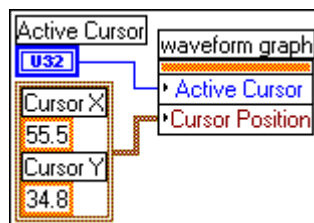


Active Cursor, Cursor Position, and Cursor Index Attributes



The attributes 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 attribute 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 attribute, select **Cursor Info»Cursor Position»All Elements**. Cursor Index accepts an integer corresponding to an element in the array (plot). To access the Cursor Index attribute, select **Cursor Info»Cursor Index**.

Wiring Example—Place a cursor at position (55.5, 34.8). When choosing 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 attributes to this node.

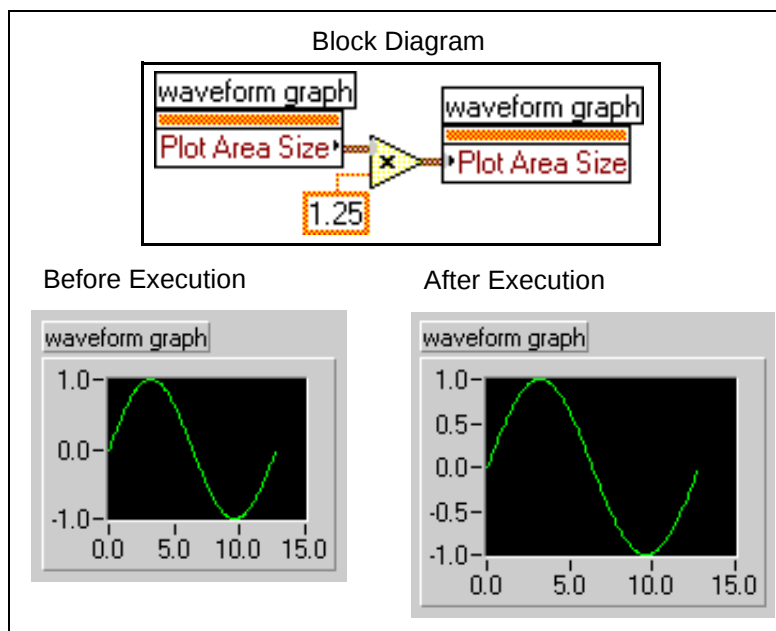


Plot Area Size Attribute



To read or change the size of a graph or chart, send new Width and Height values to the Plot Area Size attribute. 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 4-2 Acquire Data.vi

Objective: To create a VI that uses attribute nodes to clear a waveform chart and notify the user if the data exceeds a limit.

You will finish building a VI that uses attribute nodes to perform the following tasks:

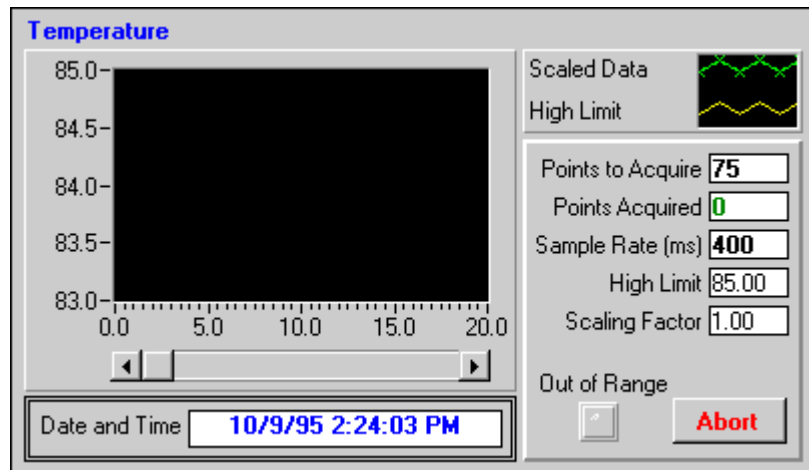
- Set the delta X value of the chart to the sample rate in seconds.
- Clear the waveform chart so it initially contains no data.
- Change the color of a plot if the data exceeds a certain value.
- Make an alarm indicator blink if the data exceeds a certain value.



Note

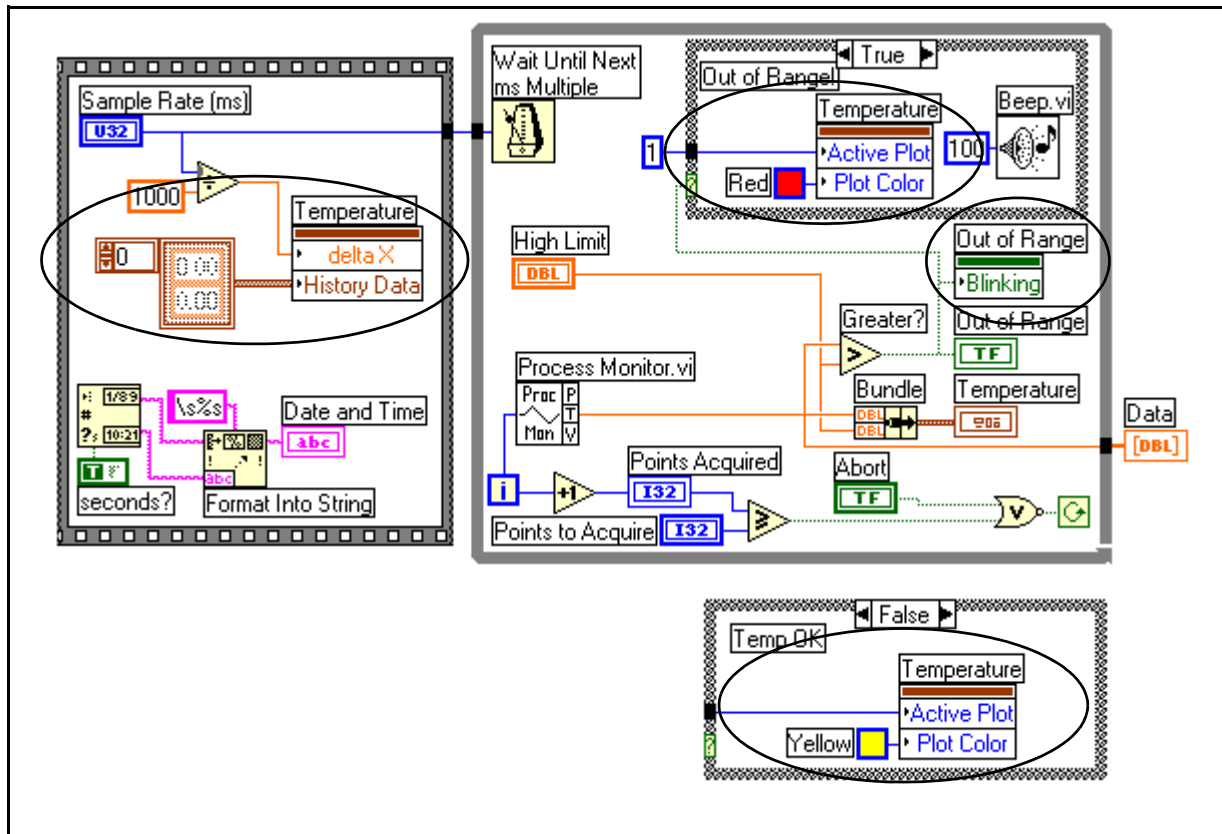
You will use this VI in the project in Lesson 6.

Front Panel



1. Open the **Acquire Data** VI in Basics2.11b. 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 of 85 degrees, and the Out of Range LED should blink.
 - a. In the Sequence structure, create an attribute node for the Temperature chart that has two terminals. One terminal should be the delta X attribute (**X Scale Info»Xo and delta X»delta X**), and the other terminal should be the History Data attribute. To clear a waveform chart from the block diagram, send an empty array of data to the History Data attribute. To do this easily, pop up on the attribute and select **Create Constant**. Make sure that the array constant is empty.
 - b. In the Case structure inside the While Loop, create an attribute node for the Temperature chart that has two terminals. The top terminal should be the Active Plot attribute, and the bottom terminal should be the Plot Color attribute (**Plot Info»Plot Color**). You will need these attributes in both cases of the Case structure. The High Limit plot is plot 1, so set the Active Plot attribute to one before setting the Plot Color attribute. If the data

is 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 attribute.

- c. Pop up on the Out of Range indicator to create the Blink attribute node. The indicator should blink when the temperature is greater than the High Limit.
2. Run the VI to confirm that it behaves correctly. Save and close the VI.

End of Exercise 4-2

Exercise 4-3 Analyze and Present Data.vi

Objective: To use attribute nodes with graph cursors.

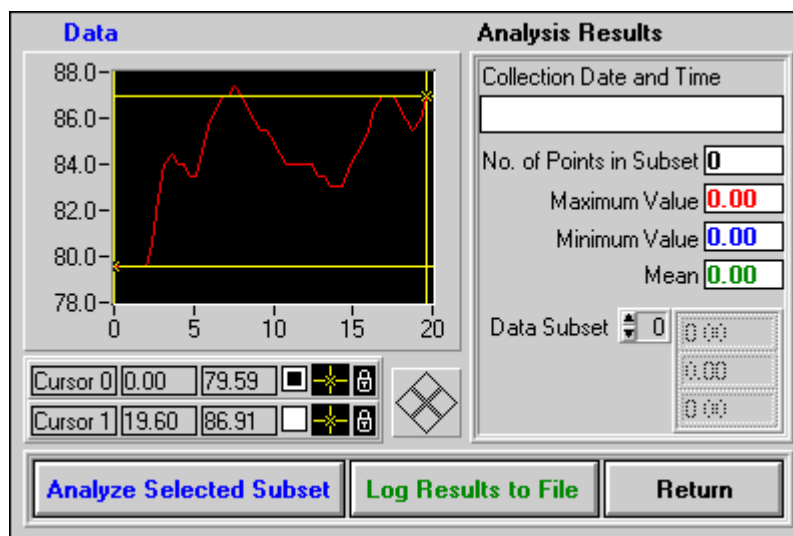
You will create a VI in which you use graph cursors to select a subset of data for analysis. In a later exercise, you will build a subVI that saves the results to disk.



Note

You will use this VI in the project in Lesson 6.

Front Panel



1. Open the **Analyze and Present Data** VI in Basics2.11b. 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.



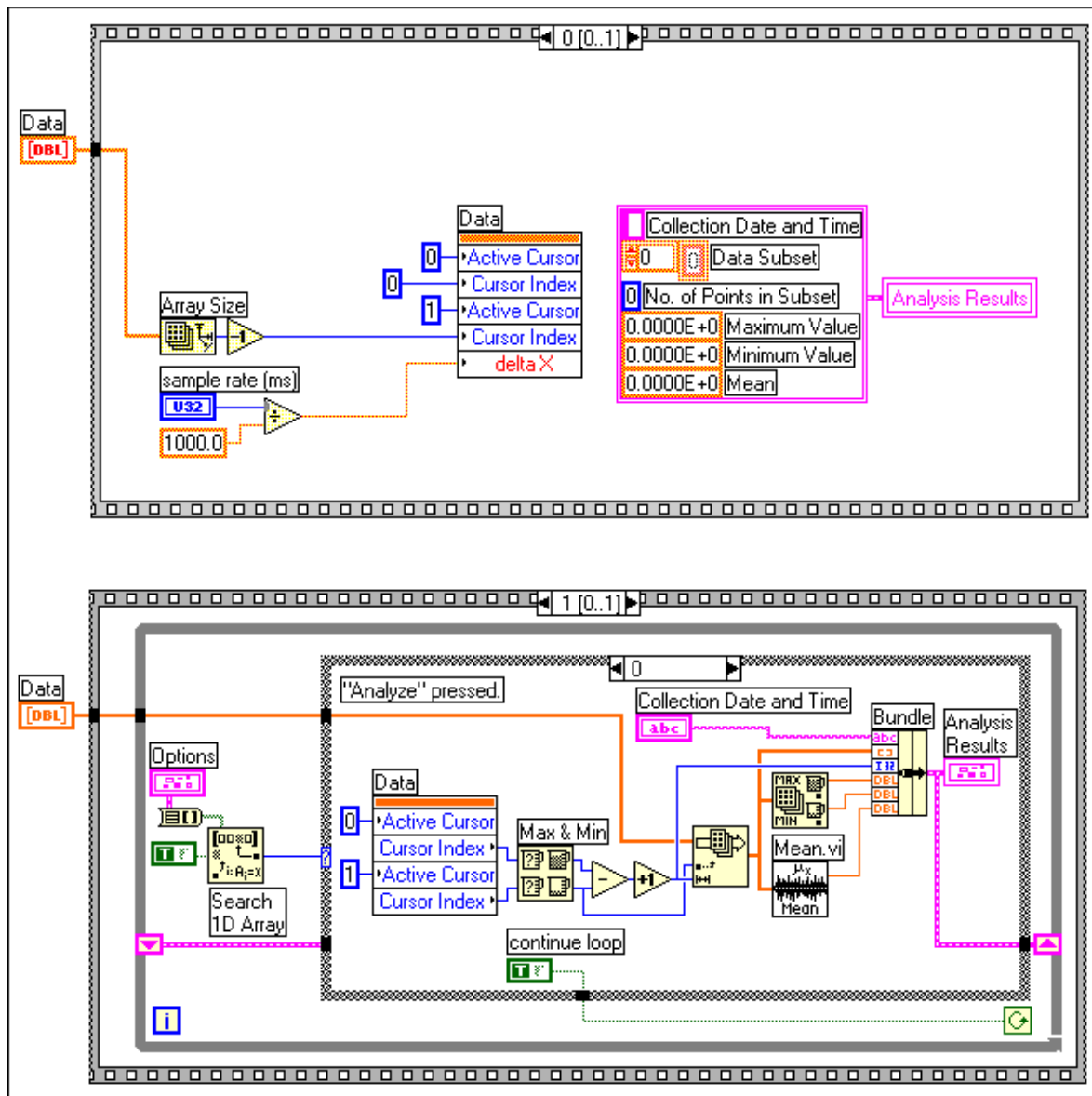
Movement locked to the plot points.



Movement unrestricted in the plot window.

In this exercise, use cursors that are locked to the plot. Use attribute 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 on the Analyze Selected Subset button, read the location of each cursor and use this information to find the average, maximum, and minimum values of the subset of data.

Block Diagram



1. Complete the block diagram shown above.

To create the attribute nodes, pop up on the Data terminal and choose **Create»Attribute Node** from the pop-up menu. Use the Positioning tool to resize the attribute node so it has enough terminals. You can then select the attribute by popping up on the newly created node and choosing the attribute from the **Select Item** menu. The Cursor Index attribute is found in **Cursor Info»Cursor Index**.



Array Subset function (**Array** palette). In this exercise, extracts the set of data points to analyze.



Max & Min function (**Comparison** palette). In this exercise, determines the first and last index for finding the subset of data to analyze.



Array Max & Min function (**Array** palette). In this exercise, returns the maximum and minimum values in the waveform.



Mean VI (**Mathematics»Probability and Statistics** palette). In this exercise, calculates the average value of the set of data elements selected by the cursors.

2. 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.
3. Save and close the VI.

End of Exercise 4-3

Summary, Tips, and Tricks

- Attribute nodes are powerful tools for expanding user interface capabilities. You use attribute nodes to programmatically manipulate the appearance and functional characteristics of front panel controls and indicators.
- All front panel controls and indicators have attributes.
- You create attribute nodes from the pop-up menu of a control or indicator (or a control or indicator terminal on the diagram).
- You can create several attribute node terminals for a single front panel object, so you can configure attributes from several locations in a block diagram.
- You can set or read attributes such as user access (enable, disable, grey out), colors, visibility, location, and blinking.
- Attribute nodes greatly expand graph functionality. Some of the many graph and chart attributes you can manipulate include the graph size, scales, and cursors.
- The Help window is a useful tool for learning more information about individual attributes from a node.

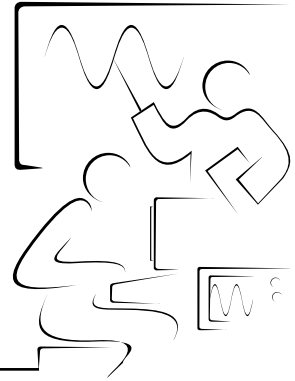
Additional Exercises

- 4-4 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 choose the following options: Enable and Show Button, Disable Button, Grey Out Button, and Hide Button. Use attribute 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.
- 4-5 Open the **Analyze and Present Data** VI you developed in Exercise 4-3. Modify the VI so that if the number of data points in the subset to be analyzed equals one, the Log Results to File button is grayed out.

Notes

Lesson 5

Advanced File I/O Techniques



Introduction

When implementing subVIs that do file I/O, you may need to account for different file formats. For example, if a third-party application will need to read data acquired by LabVIEW, you should save that data in ASCII format because most applications have support for ASCII 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 ASCII, 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 (ASCII 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 <Tab> and <Return>, 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

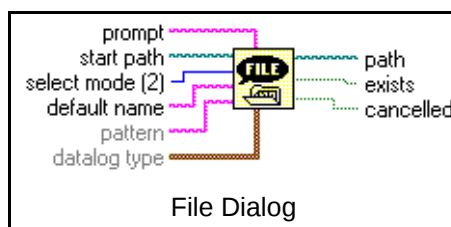
Before discussing byte stream files in detail, we will first briefly review the basic LabVIEW file I/O functions. These functions are in the **Functions»File I/O** and **Functions»File I/O»Advanced File Functions** palettes.



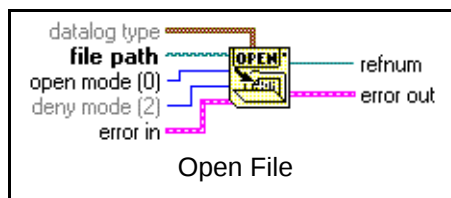
Note

Use the Online Reference (Help menu) for more information on these functions.

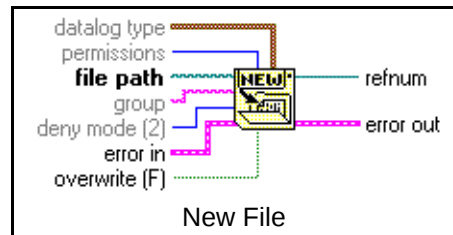
The **File Dialog** function (**File I/O»Advanced File Functions** palette) displays a file dialog box for file selection. You can select new or existing files or directories from this dialog box. We will discuss the use of the **datalog type** input later in this chapter.



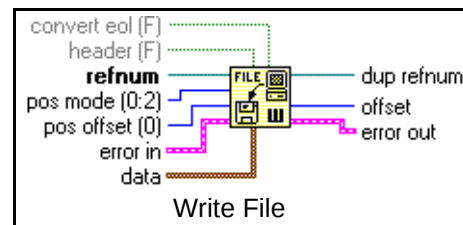
The **Open File** function (**File I/O»Advanced File Functions** palette) opens an existing file. You must wire a valid path to the **file path** input. This function is *not* capable of creating or replacing files. It opens only *existing* files. The **datalog type** input is used only when opening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



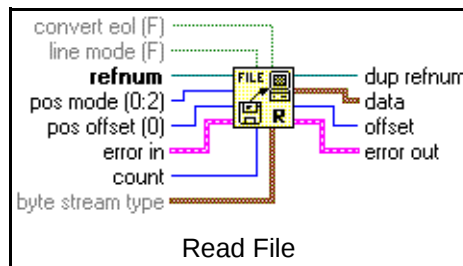
The **New File** function (**File I/O»Advanced File Functions** palette) creates a **new file** for reading or writing. You must wire a path to the file path input. The **datalog type** input is used only when creating new LabVIEW datalog files.



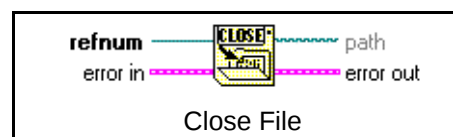
The **Write File** function (**File I/O** palette) writes data to an open file. The behavior of this function varies slightly depending on whether you are writing data to a byte stream file or a LabVIEW datalog file.



The **Read File** function (**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 discuss this concept later in this lesson.

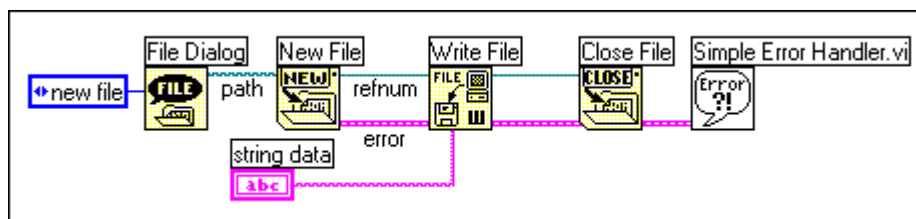


The **Close File** function (**File I/O** palette) closes the file associated with **refnum**.



Byte Stream Files

You can use the **Write File** function to create both types of byte stream files—ASCII and binary. Creating an ASCII file is as simple as sending a string containing characters to the **Write File** function.



When to Use ASCII Files

ASCII files are useful and common for many reasons. Almost every operating system and a majority of software applications can read and write files in ASCII format. Historically, most instrument control applications (such as GPIB and serial) use ASCII strings to send control statements. With the advent of message-based VXI instrumentation and the VISA instrument architecture, the continued use of ASCII-based information and data exchange seems certain.

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

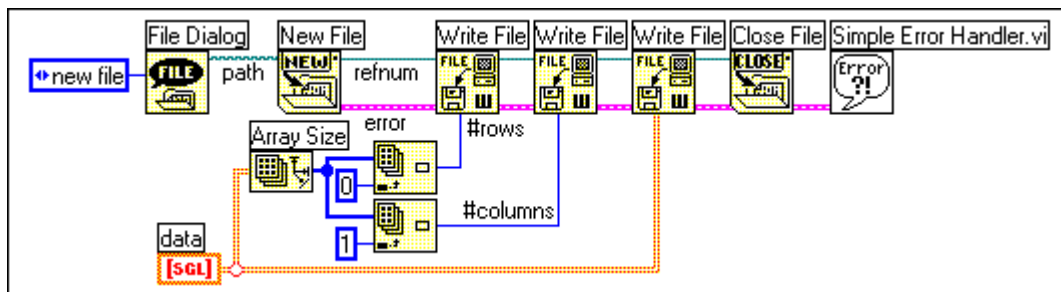
ASCII files also have some significant disadvantages. First, it is difficult to randomly access data in an ASCII file. Even though each character in the string takes up one byte of space, there is no standard width for numeric fields. In addition, ASCII files can require significantly more disk space than other file formats to store information. Thus, using ASCII files in your application may degrade performance in some situations.

As shown above, the **Write File** function skips over any header information that LabVIEW uses to store the string in memory and simply writes the contents of the string data control to the file. (For a more detailed discussion of data types, see section A of Lesson 8, *Additional Topics*.) In fact, the **Write File** function does not distinguish an ASCII string from a binary string when writing the data to the file—it simply 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 simply 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.

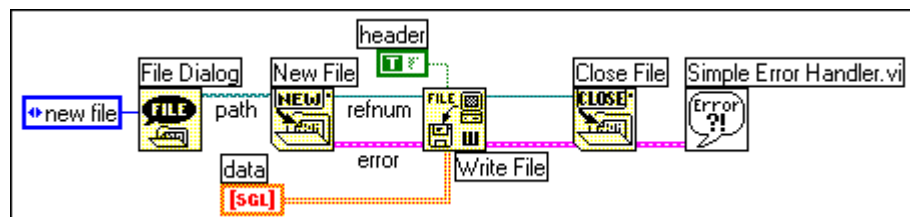
For example, if you wire a two-dimensional array of numbers into the **data** terminal, the **Write File** function places a byte-for-byte copy of the input data into the file. It does not convert the numbers to ASCII, nor does it place any information about the number of rows or columns in the array into the file. Even if you know that the data was 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.

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 the number of columns of data.



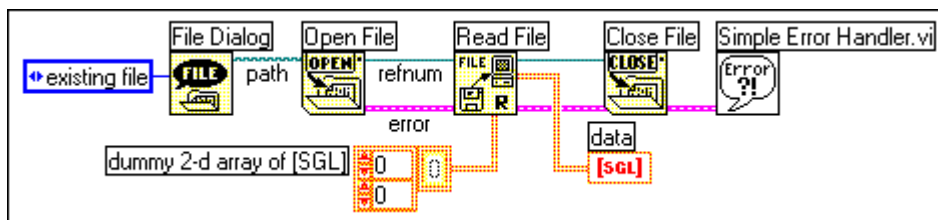
You can also generate the same file using the **header** input of the **Write File** function. If you wire a Boolean value of TRUE to the **header** input of the **Write File** function, it writes the same data to the file as if you had manually written a header. The following example shows how to use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



Using the above 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 as well as 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 below, you can read the binary file from the previous examples by using the **Read File** function.

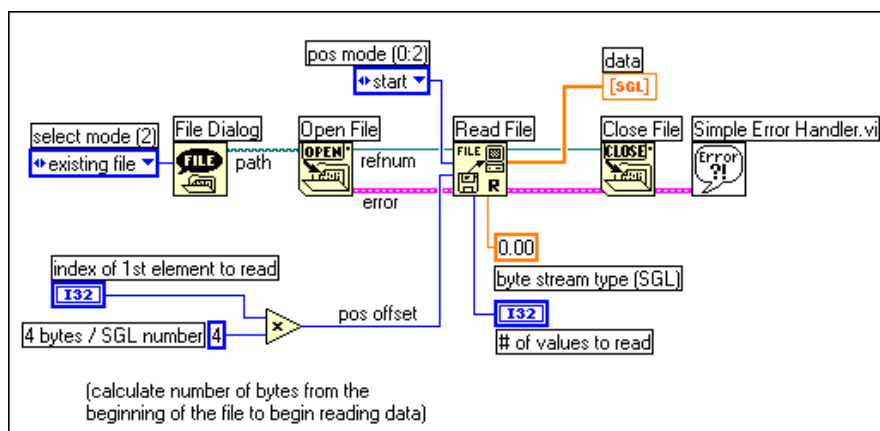


Notice that the **byte stream type** input of the **Read File** function has a two-dimensional array of single-precision numbers wired to it. The **Read File** function uses only the *data type* of this input to read the data in the file, assuming that it has the correct header information for that data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

Random Access in Byte Stream Files

If you store arrays of numeric data in a file, you may find it necessary to access data at random locations in the file. Random access of data stored in ASCII files is hampered by the presence of negative signs, varying number of digits in individual data points, and other factors. For example, given 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 ASCII file. The problem is that in ASCII, 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 easily read an arbitrary group of elements from the array, as shown below.



In the above 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 $4 * n$ 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, the output data is placed in an array, because more than one value can be read from the file.

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

- When performing ASCII and binary file I/O, remember that values for the **pos offset** terminal are measured in bytes.
- When using the **Read File** function, the **count** input 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.
- You can find more detailed information in the **Online Reference** and the *LabVIEW Function Reference Manual*.

When to Use Binary Files

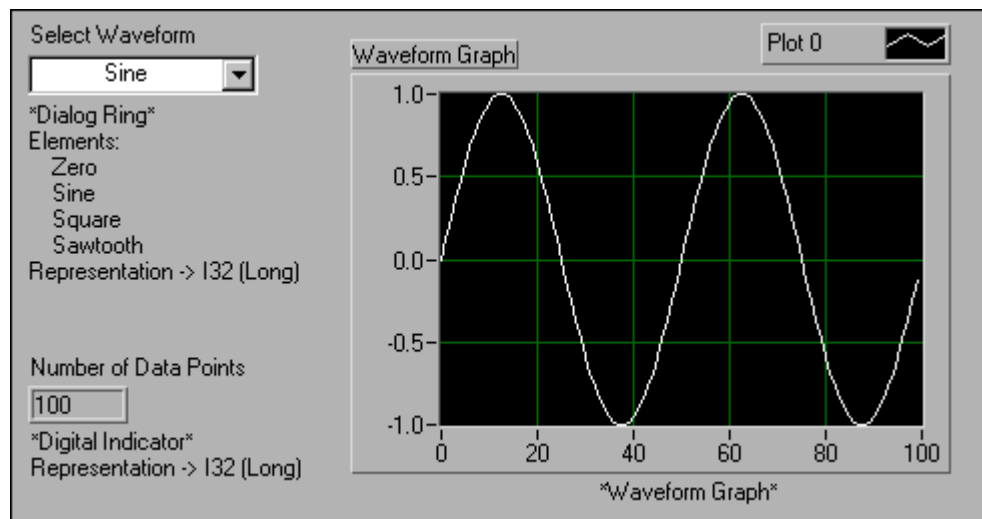
Binary files can be much more efficient than ASCII files because they typically require less disk space to store the same amount of information. When numeric data is stored in a binary file, it is also relatively simple to randomly access data stored in the file. However, binary files are generally not portable to other applications. Also, you must carefully document the details needed to extract information stored in a binary file.

Exercise 5-1 Binary File Writer.vi

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

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, you will build a VI that reads the binary file.

Front Panel



1. Open a new VI.
2. Build the front panel shown above. When you create the dialog ring, recall that a shortcut for adding a new item to the list of options is to press <Shift + Return> after entering the text for an item in the list. Item 0 in the list should have the text Zero, item one should have the text Sine, and so on. To view the numeric value of the dialog ring, pop up on the control and select **Show»Digital Display**.

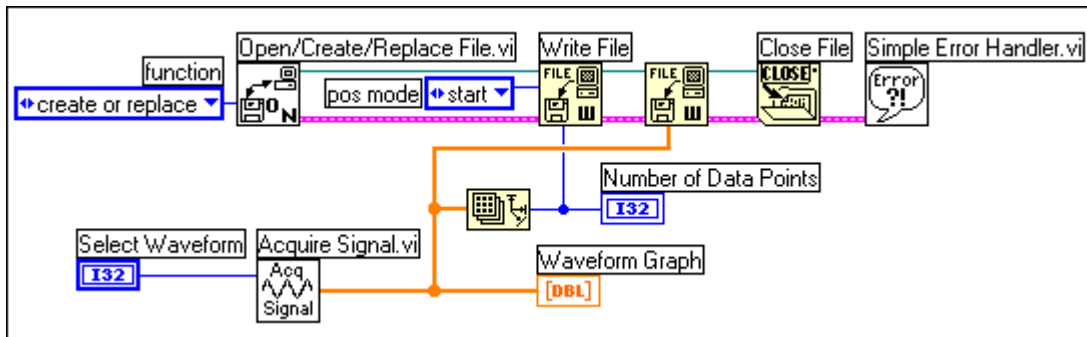


Note

The appearance of the dialog ring control will vary from one platform to the next.

3. Show the block diagram.

Block Diagram



1. Build the block diagram shown above.



Open/Create/Replace File VI (File I/O palette). This VI creates or replaces a file.



Write File function (File I/O palette). 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.



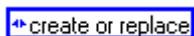
Close File function (File I/O palette). In this exercise, closes the binary file after data has been written to it.



Simple Error Handler VI (Time & Dialog palette). In the event of an error, this VI displays a dialog box with information about the error and where it occurred.



To create this constant, pop up on the **pos mode** input of the **Write File** function and select **Create Constant**. Setting the position mode to start ensures that new data is written relative to the beginning of the file.



To create this constant, pop up on the **Function** input of the **Open/Create/Replace File VI** and select **Create Constant**. By selecting **create or replace**, you are allowing the user to create a new file or overwrite an existing file.



Acquire Signal VI (User Libraries»Basics-II Course palette). In this exercise, generates the waveform selected by the Select Waveform control.



Array Size function (Array palette). In this exercise, returns the number of elements in the 1D array of data to be written to the file.

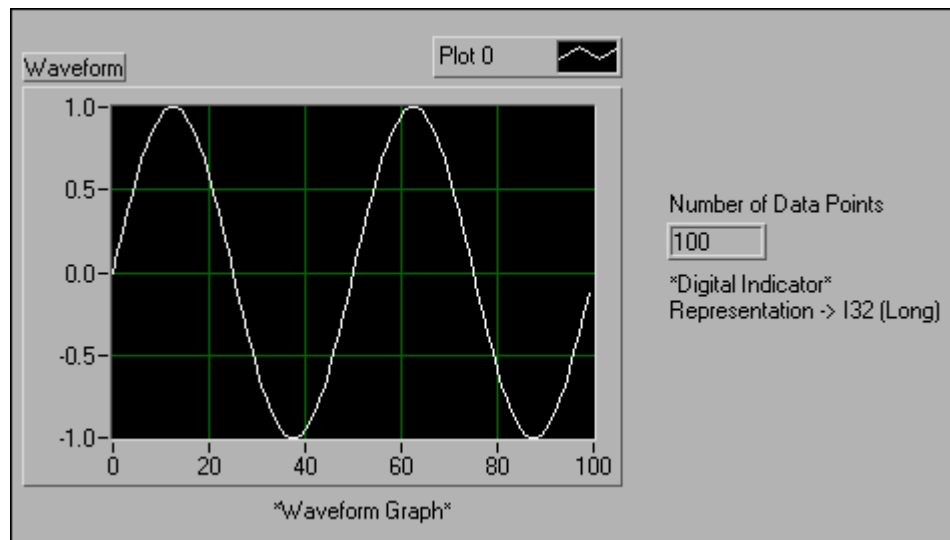
2. Save the VI as **Binary File Writer.vi**. Run it, saving data to the file on disk named `data.bin`.

End of Exercise 5-1

Exercise 5-2 Binary File Reader.vi

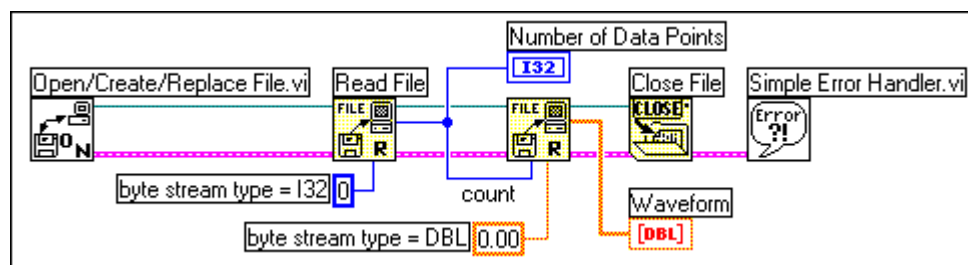
Objective: To build a VI that reads the binary file created in the last exercise.

Front Panel



1. Open a new VI and build the front panel shown above.
2. Show the block diagram.

Block Diagram



1. Build the block diagram shown above.



Open/Create/Replace File VI (File I/O palette). This VI opens a file.



Read File function (File I/O palette). 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.



Close File function (**File I/O** palette). In this exercise, this function closes the binary file after data has been read from it.



Simple Error Handler VI (**Time & Dialog** palette). In the event of an error, this VI displays a dialog box with information about the error and where it occurred.

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

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 5-1.

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

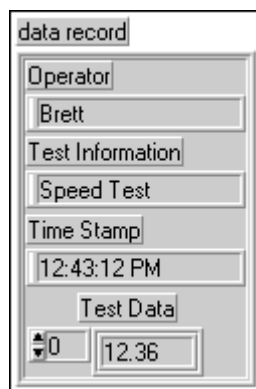
4. Close the VI.

End of Exercise 5-2

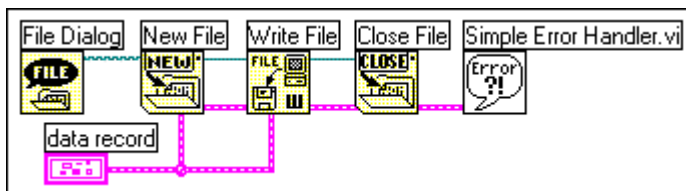
B. LabVIEW Datalog Files

If your data has a mixture of data types, formatting it into ASCII or binary strings to store it may 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 may 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 example at the right, 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.

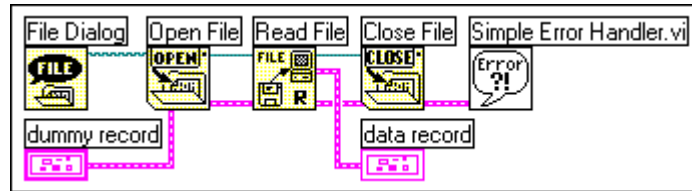


You use the same file I/O functions to work with datalog files as you use for byte stream files. However, you use the data type input differently for datalog files. To create a new datalog file, you wire a cluster matching the data record cluster to the **datalog type** terminal of the **New File** function. This cluster specifies how the data will be written to the file. Then you 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 back 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** will return an array of records.

When to Use Datalog Files

You can use datalog files to store and retrieve complex data formats quickly and easily 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 quickly and easily.

Exercise 5-3 Save Data to File.vi

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

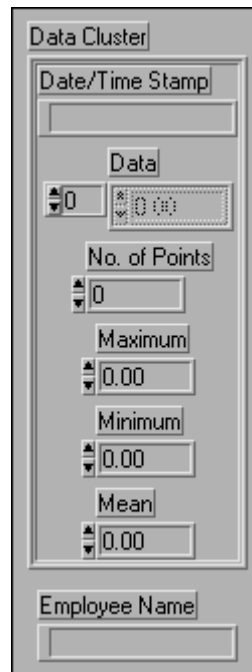


Note

Completing this VI will enable the Log Results to File option in the Analyze and Present Data VI from Exercise 4-3. Recall that you will use Exercise 4-3 in the project in Lesson 6.

In the application being developed, a mixed data set needs to be saved to disk. This data set contains simple numerics, arrays of numerics, and string data. In addition, this data will be used only from within LabVIEW. Because of these requirements, datalog files provide an easy way to save the application's data subsets to disk.

Front Panel

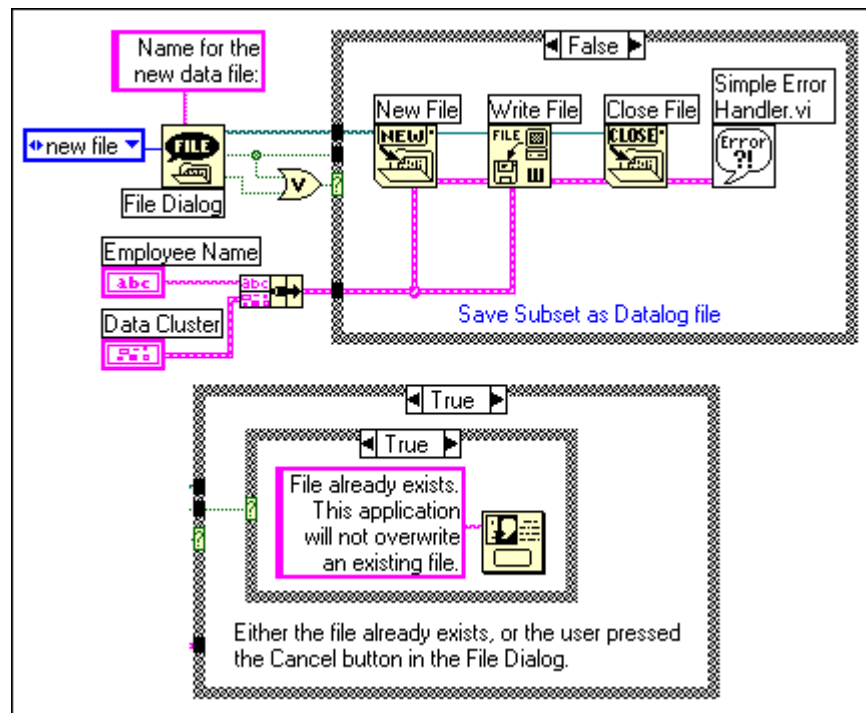


1. Open the **Save Data to File** VI in Basics2.11b. The front panel is already built.

The two controls on the front panel will be used to pass application data to this VI, which will function 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



1. Complete the Case structures for the block diagram as shown above. The cases that are not shown are already built.
 - a. Delete the Boolean constant labeled **Delete Me!**. Create the **File Dialog** function, connecting the output of the **exists** and **cancelled** terminals to the inputs of the **OR** function. The **exists** value should also be sent to the Case structure as shown in the figure above.



File Dialog function (**File I/O»Advanced File Functions** palette). This function prompts for the name of the new file.



To create this constant, pop up on the **select mode** input of the **File Dialog** function and select **Create Constant**. Set this constant to the value **new file** with the Operating tool.

Name for the new data file:

To create this constant, pop up on the **prompt** input of the **File Dialog** function and select **Create Constant**. This string displays a prompt message in the file dialog box.

- b. 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.



New File function (**File I/O»Advanced File Functions** palette). This function creates the new file. Notice that the **datalog type** input to this function must receive an input.



Write File function (**File I/O** palette). This function simply writes a record to a datalog file.



Close File function (**File I/O** palette). In this exercise, this function closes the file after the data is written to it.



Simple Error Handler VI (**Time & Dialog** palette). In the event of an error, this VI displays a dialog box with information about the error and where it occurred.

2. After finishing this VI, save and close it.
3. Open **Analyze and Present Data.vi**, which you completed in Exercise 4-3. This VI calls **Save Data to File**. When you run this VI and press the Log Results to File button, the **Save Data to File** VI should pop up a dialog box so you can name the data file to save. Once the filename is selected, the data set is saved as a datalog file.

End of Exercise 5-3

Exercise 5-4 View Analysis File.vi

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

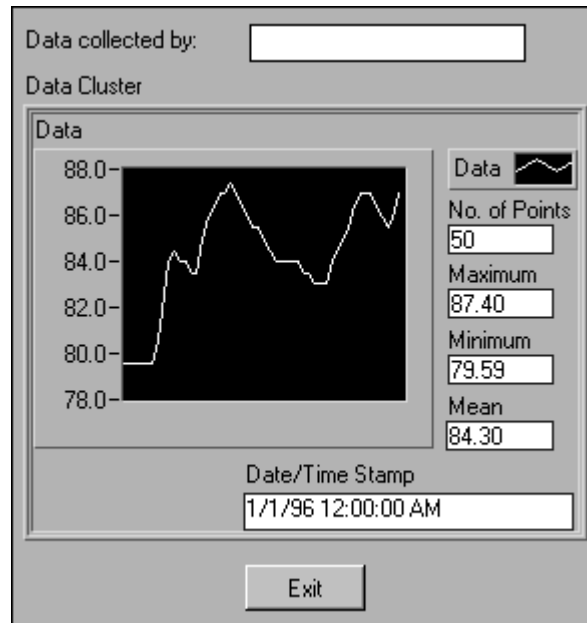
This VI reads and displays the data stored by the **Save Data to File VI**.



Note

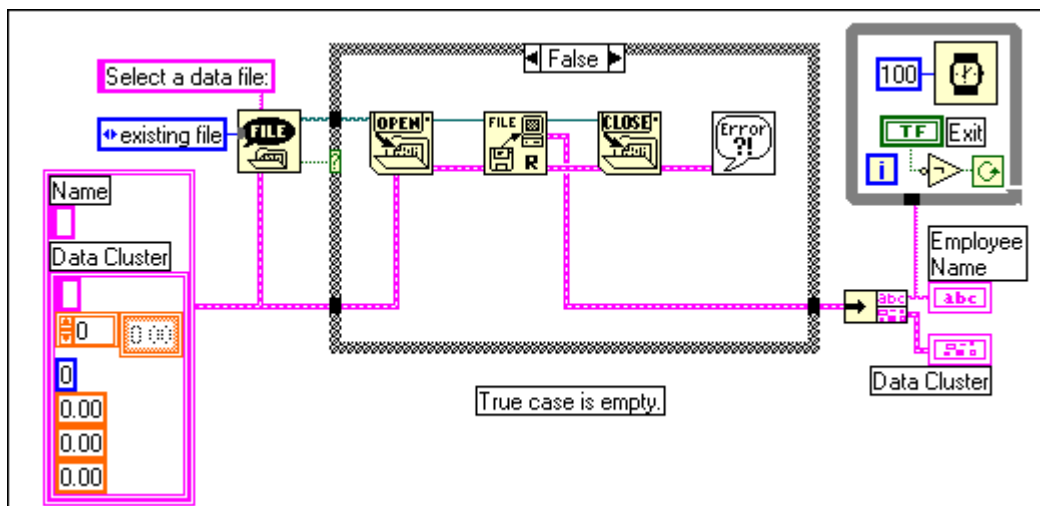
You will use this VI in the project in Lesson 6.

Front Panel



1. Open the **View Analysis File VI** in Basics2.11b. This VI has already been built.
2. Show the block diagram.

Block Diagram



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

Notice that the File Dialog function's datalog type input is wired to a dummy cluster of the same datatype 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.

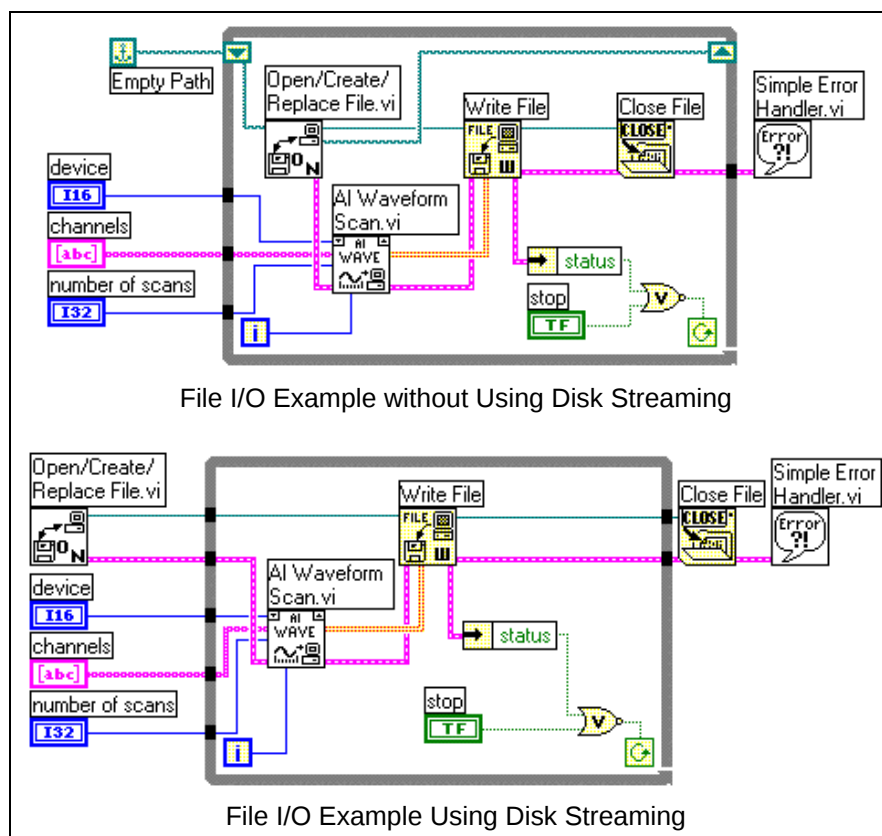
2. After you have run the VI and tested it, close it. Do not save any changes.

End of Exercise 5-4

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 (see Exercise 5-3 for one example) without discussing its performance advantages.

The examples below 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 execute. Thus, if you are calling one of these VIs in a loop, you are performing 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 ASCII, binary, or datalog files. The same basic operations of **Open File**, **Read File**, **Write File**, and **Close File** work with all types of files.

ASCII Files

ASCII data files are files in which all data is stored as readable ASCII characters. ASCII data files are useful because almost all software applications and operating systems can read them. However, ASCII files can be larger than necessary and therefore, slower to access. It is also very difficult to perform random access file I/O with ASCII files. You typically use ASCII 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 ASCII 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 will 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 quickly and easily. 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 execute, incurring unnecessary overhead in each iteration of the loop.

Additional Exercises

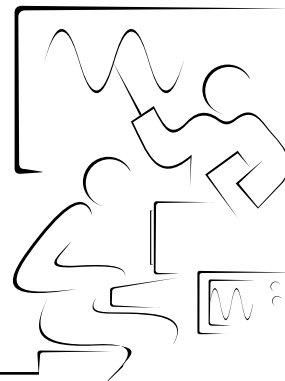
- 5-5 Write a VI that uses the Advanced file I/O functions to create a new file. Then, write to that file a single string composed of a string input by the user concatenated with a number converted to an ASCII string using the **Format Into String** function. Name the VI **File Writer.vi**.
- 5-6 Write a VI that uses the Advanced file I/O functions to read the file created in Exercise 5-5. Name the VI **File Reader.vi**.

Hint: Use the EOF function (**File I/O»Advanced File Functions** palette) to obtain the length of the written file.

Notes

Lesson 6

Developing Larger Projects in LabVIEW



Introduction

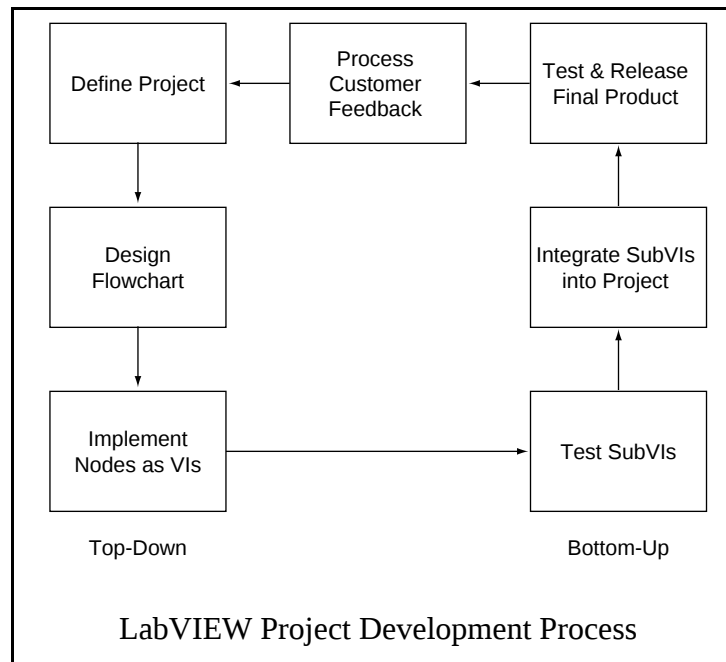
This lesson discusses some of the issues involved when building larger LabVIEW projects, including the design process, the organization of subVI components, and the process of combining those components to create a complete application.

You Will Learn:

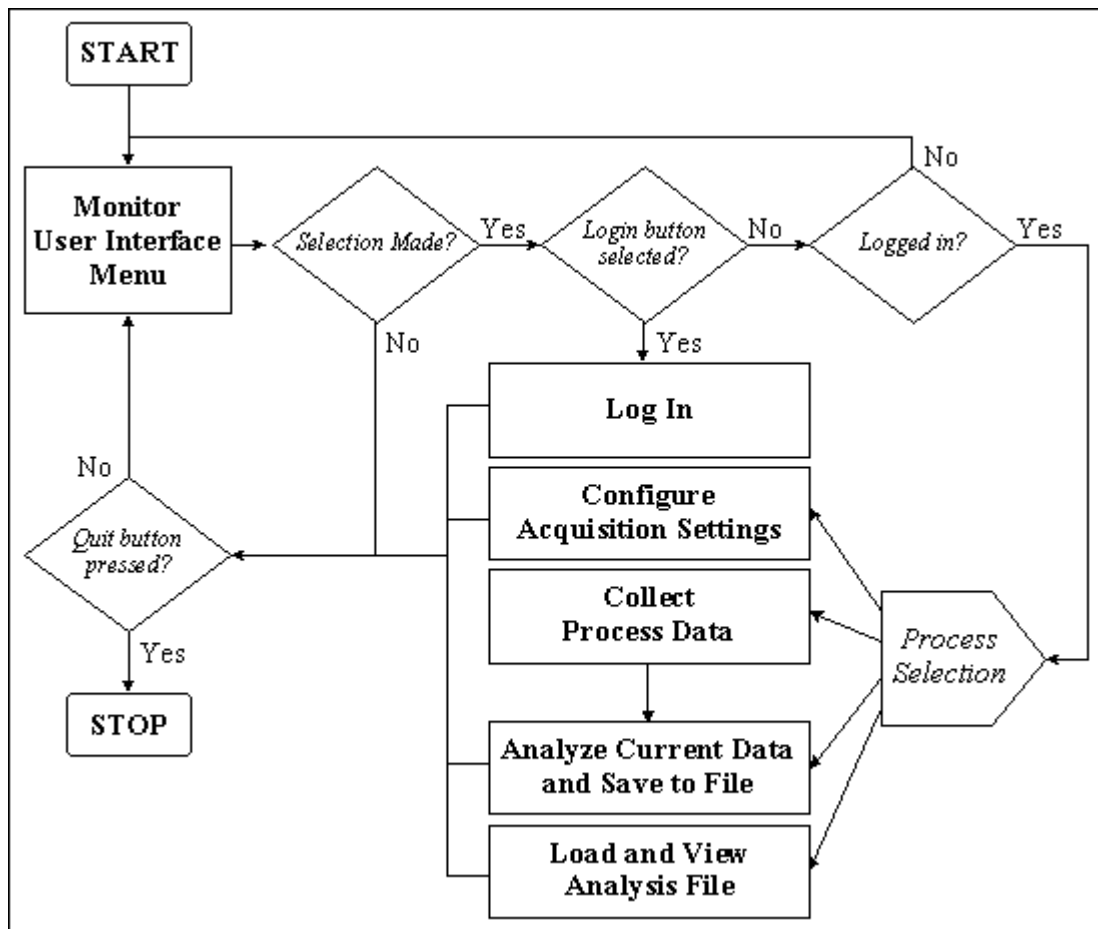
- A. How to assemble your LabVIEW application from developed subVIs.
- B. The LabVIEW features for managing project development.
- C. About LabVIEW tools for project management.

A. Assembling a LabVIEW Application

Lesson 1 of this course discussed 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. This development cycle is reviewed below.



In lessons 2 through 5, you concentrated on the bottom elements of this process, implementing and testing subVIs which correspond to nodes of the project flowchart. This project flowchart is shown below:



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 History window 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 LabVIEW VIs and LLBs (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 in the VI Information window. You should also indicate that you have checked out the VI by adding a comment in the VI History window. When you lock a VI, its 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.

Typing information about a VI in the VI Information window and using the Description entry of front panel controls and indicators are easy ways to clearly document your VIs. Control descriptions also provide online help you can use in the final product.

Using LabVIEW Libraries (LLBs)

LabVIEW library files (.LLB files) are a useful means to organize 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 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.

- When LLB files become large (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.

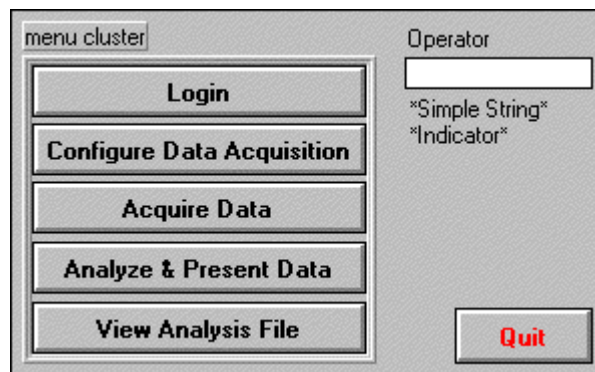
As a rule of thumb, try to limit the size of your LLB to roughly 1 MB. Whether you store your VIs in an LLB or directly on disk may depend on file naming restrictions in your operating system or how you choose to organize your files.

Exercise 6-1 Application Exercise (6-1).vi

Objective: To use the Login VI created in Exercise 4-1 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 Basics2.llb. You created this VI in Exercise 2-3.

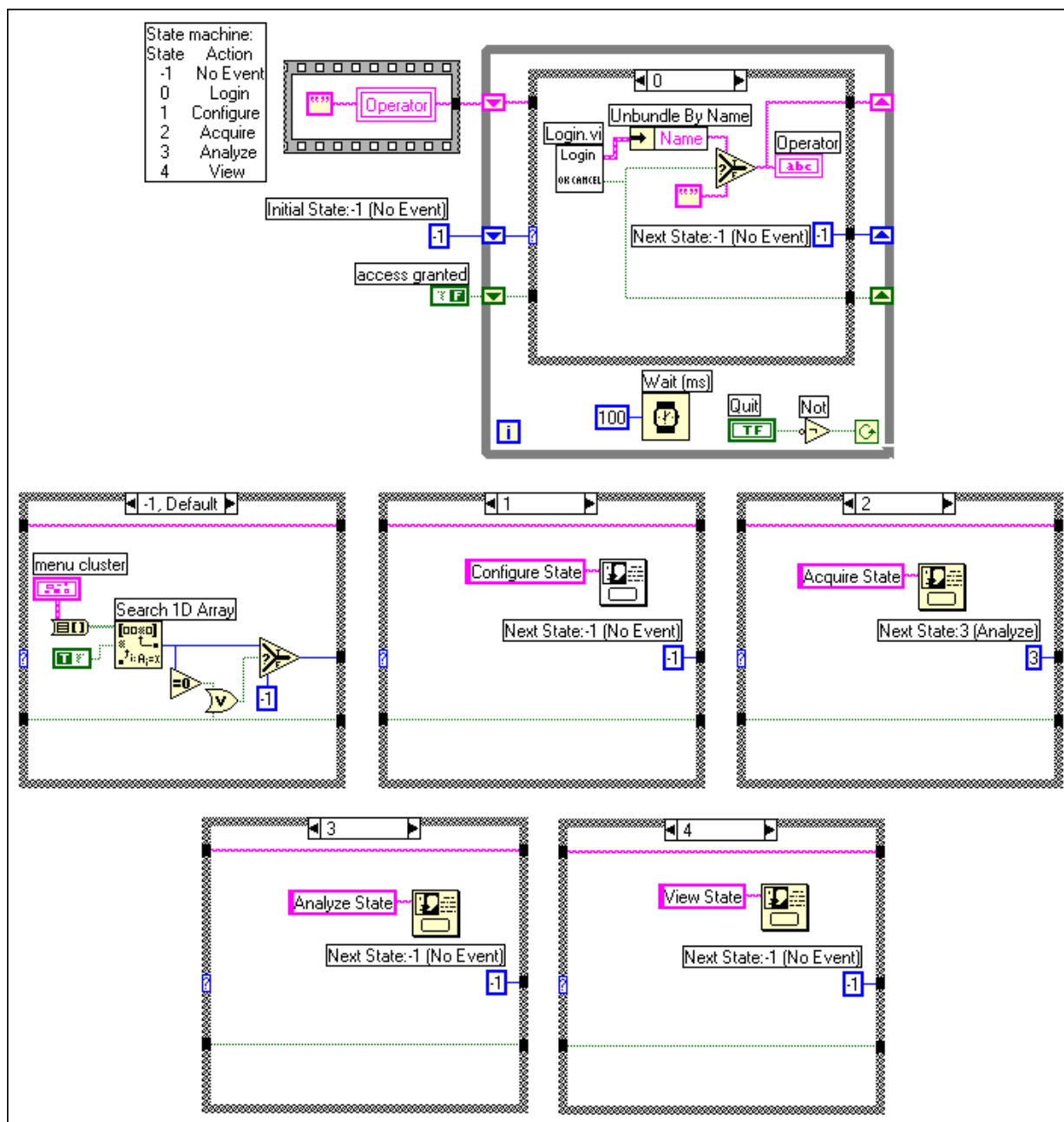


Note

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

2. Add a Simple String indicator (**String & Table** palette) to the front panel. Label it Operator.
3. Open the block diagram.

Block Diagram



1. Modify the VI so that it calls the **Login** VI 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 4-1, 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 menu option should 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 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, as shown in the figures on the previous page.

By using the shift registers, notice that only Case 0, in which the **Login** VI executes, can change the Operator and access granted values.

The loop checks to see if 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) to determine which case to execute. If both 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(6-1).vi**. Do not close the VI.

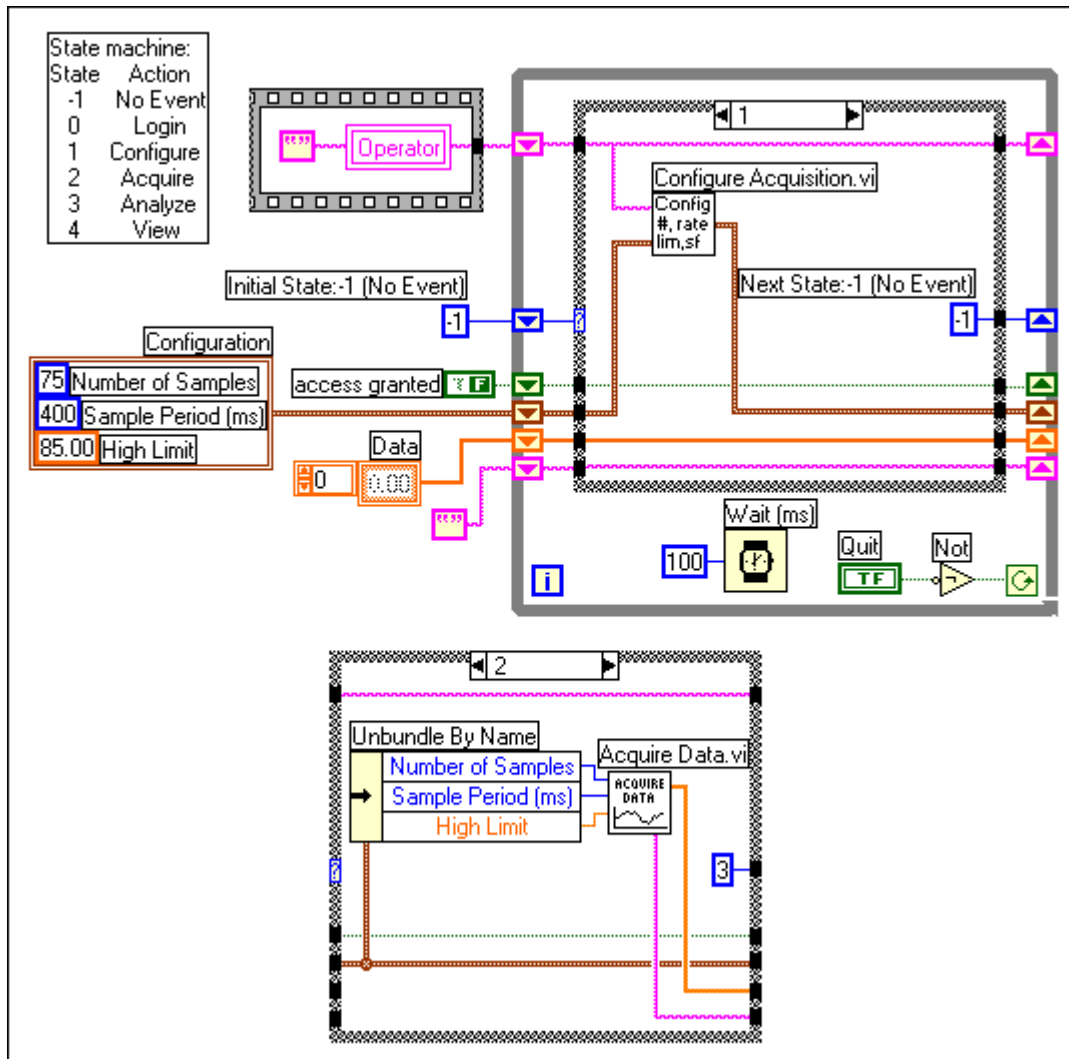
End of Exercise 6-1

Exercise 6-2 Application Exercise (6-2).vi

Objective: To add the **Configure Acquisition** and **Acquire Data** functions to the application started in Exercise 6-1.

In the previous exercise, you began building the **Application Exercise VI**. Now, you will add the **Configure Acquisition VI**, which has been previously built, and the **Acquire Data VI**, which you finished in Exercise 4-2.

Block Diagram



1. Make sure that the **Application Exercise(6-1)** 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 1 of the Case structure. Add the **Configure Acquisition** VI to this case. You can find this VI in **Functions»User Libraries»Basics-II Course**.
 - a. Connect the string containing the operator name to the Operator input.
 - b. Add a shift register to the border of the loop. Connect the Configuration out terminal of **Configure Acquisition** to the right side of the shift register.
 - c. Connect the value from the left side of the shift register you created in step 2b to the Configuration in terminal of the **Configure Acquisition** VI.
3. Show Case 2 of the Case structure and delete the **One Button Dialog** function. Add **Acquire Data.vi**, which you finished in Exercise 4-2. If you did not finish that exercise, use the VI from the solutions located on your computer.
 - a. Use the **Unbundle By Name** function (**Cluster** subpalette) to unbundle the Number of Samples, Sample Period (ms), and High Limit items from the cluster containing the configuration information. Recall that you placed this cluster of information into a shift register.
 - b. Add a shift register to the border of the While Loop. Connect the Data output of the **Acquire Data** VI to the right side of the shift register.
 - c. Pop up on the loop border to add another shift register. Wire the Date and Time output of **Acquire Data.vi** to the right side of the shift register.
4. You must initialize the new shift registers you have created in this exercise.
 - a. Pop up on the left side of the shift register that contains the configuration cluster and choose **Create Constant** from the pop-up menu. Set the Number of Samples to 75, Sample Period (ms) to 400, and High Limit to 85.0. This will be the default configuration for acquiring data until the user changes it by pressing the Configure Data Acquisition button.
 - b. Pop up on the left side of the shift register storing the Data output from **Acquire Data.vi** and choose **Create Constant** from the pop-up menu. Initially, this shift register will contain an empty array.
 - c. Wire an **Empty String** (**String** subpalette) to the left side of the shift register that stores the Date and Time output from **Acquire Data.vi**.

5. 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 passes through the other cases unchanged (recall the method used in Exercise 6-1).
6. Save the VI as **Application Exercise(6-2).vi**. Run it and test it.

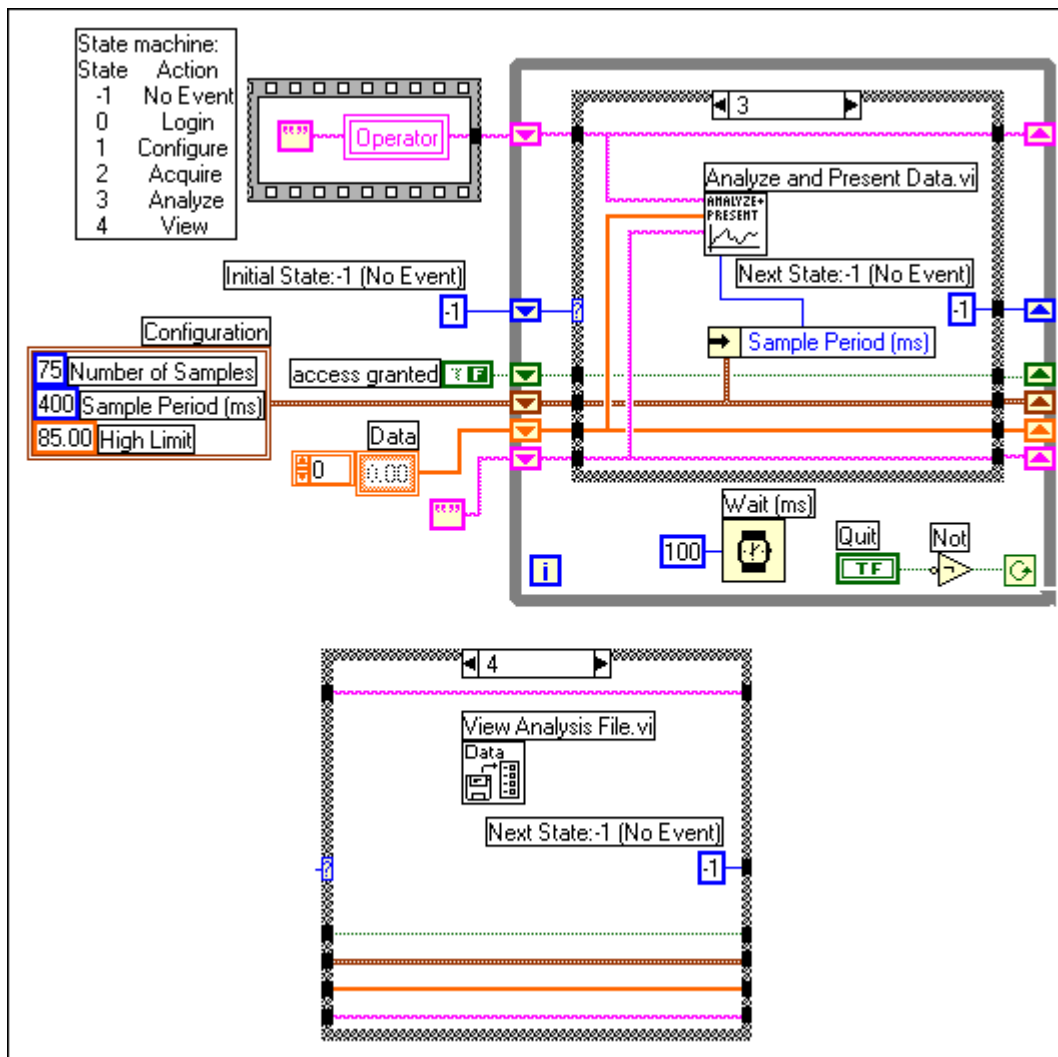
End of Exercise 6-2

Exercise 6-3 Application Exercise (6-3).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 and Present Data** VI from Exercise 4-3 and the **View Analysis File** VI, which you studied in Exercise 5-4.

Block Diagram



1. Make sure that the **Application Exercise(6-2)** 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 3 of the Case structure. Add the **Analyze and Present Data** VI to this case. You completed this VI in Exercise 4-3.
 - a. Connect the string containing the operator name to the Operator input.

- b. Connect the array containing the collected data to the Data input.
 - c. Connect the string containing the date and time to the Collection Date and Time input.
 - d. Use the **Unbundle By Name** function to unbundle the Sample Period (ms) item from the cluster wire containing configuration information. Wire this value to the sample rate (ms) input of **Analyze and Present Data.vi**.
3. Show Case 4 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(6-3).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, press the **Analyze** button and then select the button to save the data to file. Otherwise, the file will not contain any waveform data.*

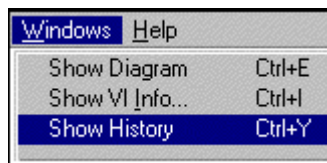
End of Exercise 6-3

B. LabVIEW Features for Project Development

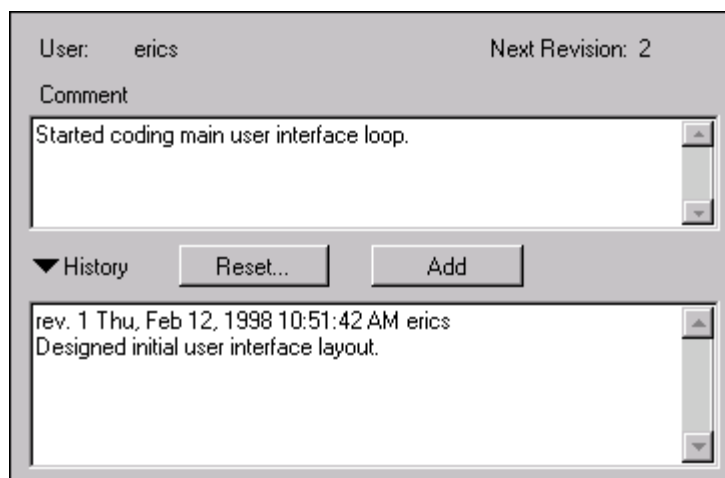
VI History

One of the most useful LabVIEW tools for team-oriented development is the VI 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 History window to track changes and revisions to VIs and applications. You can configure LabVIEW with several different VI History options, which apply on a global or per-VI basis (using the LabVIEW preferences or **VI Setup**).

To open the History window for a VI, choose **Show History** from the **Windows** menu.



The History window appears as shown below.



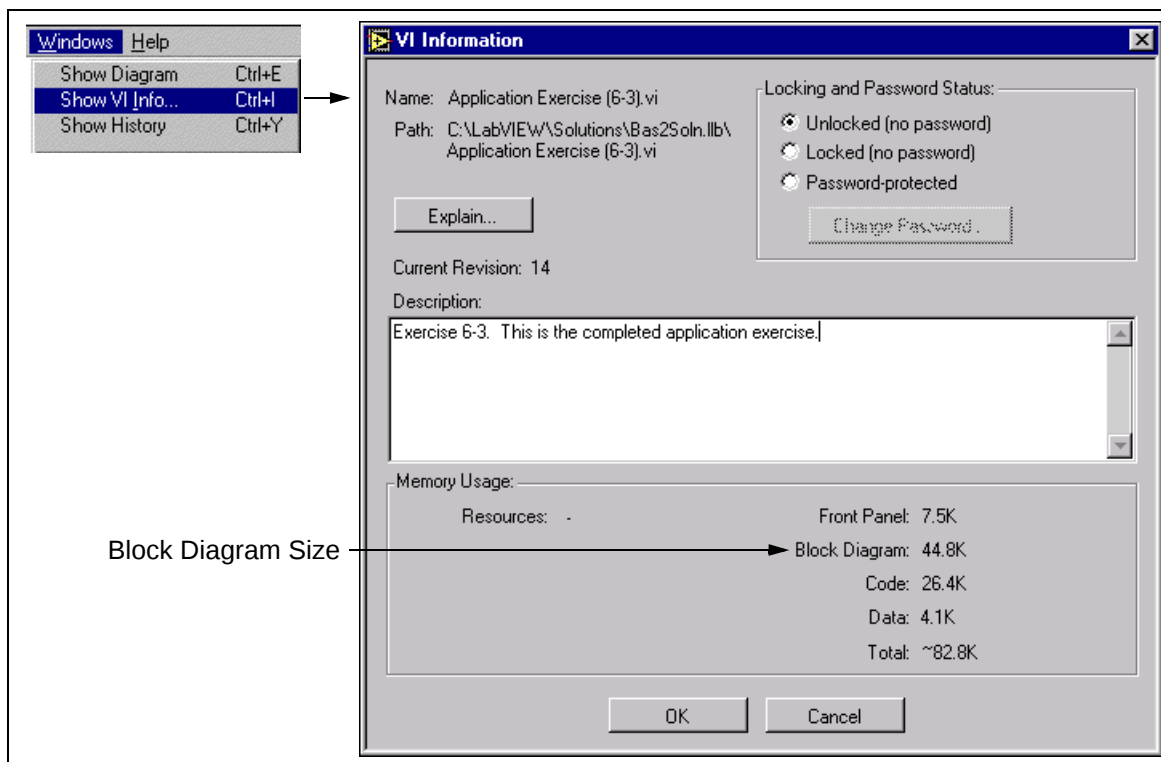
After typing your comments into the Comment area of the History window, click on **Add** to add them to the VI's history. The VI History window also keeps track of revision numbers. Each time you save the VI, the revision number for the VI increments by one.

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, or you can change the user name by selecting **User Name** from the **Edit** menu.

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, we recommend 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, choose the **Show VI Info** option from the **Windows** menu. 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.

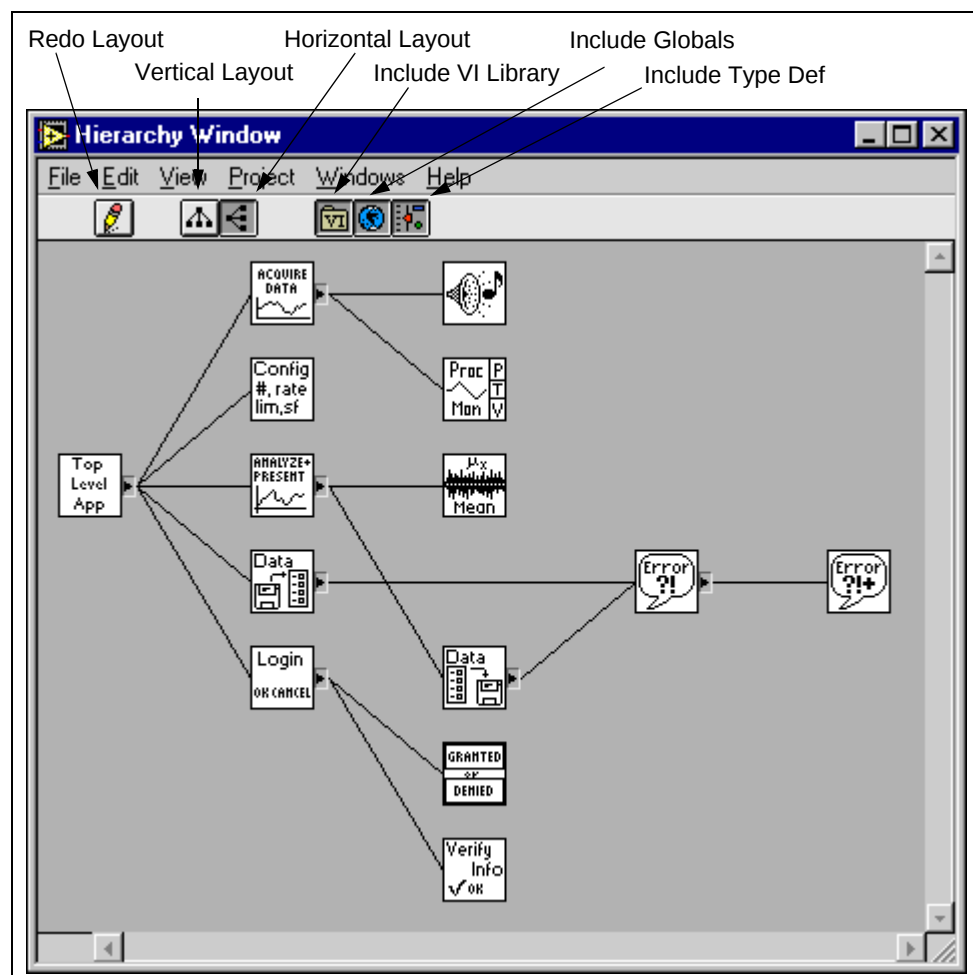


If you find that the block diagram for a VI is getting too large, you can easily convert part of it into a subVI by using **SubVI From Selection** in the **Edit** menu. This capability gives you a fast and easy method to create your application's VI hierarchy as you develop the source code.

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 **Show VI Hierarchy** from the **Project** menu. 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 on 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 as you see fit.

The Hierarchy window shown below contains the hierarchy of the application you completed in the previous exercise. The entire hierarchy was shown by popping up on 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 appears below the VI. You can double-click on 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.

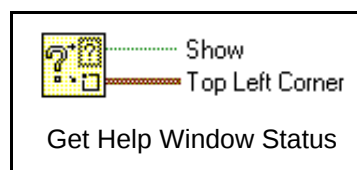
The Hierarchy window can also be used a development tool when planning or implementing your project. For example, after developing a flowchart of the VIs required for an application, you can create (from the bottom of the hierarchy up) each of these VIs so that they have all necessary inputs and outputs 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

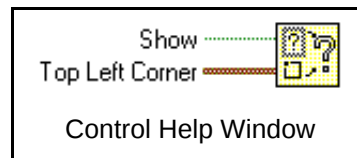
As you put the finishing touches on your application, you may want to provide online help to the user. LabVIEW offers several mechanisms for doing this. By using the **Description** option available on every front panel control and indicator, you create not only well-documented VIs, but also VIs that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

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

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



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

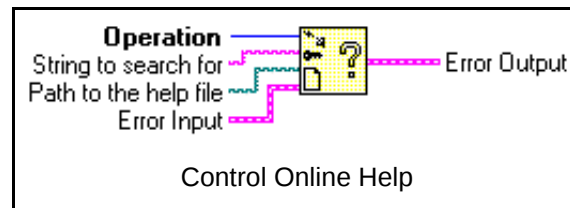


You can also use the **Control Online Help** function to access LabVIEW's Online Reference or custom help files that you compile using third-party tools. The type of online reference development tools you can use to develop

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

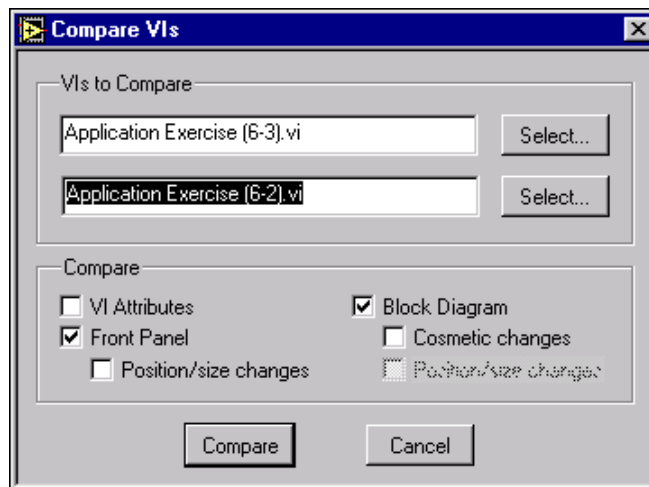
With this function, you manipulate an online help file. You can list the contents of the help file, jump to keywords in the file, or close the online help file.



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 G Developers toolkit (part of the LabVIEW Professional Developers Suite) includes a utility to determine the differences between two VIs loaded into the memory. From the LabVIEW pull-down menu, select **Project»Compare VIs...** to bring up the Compare VIs dialog box:



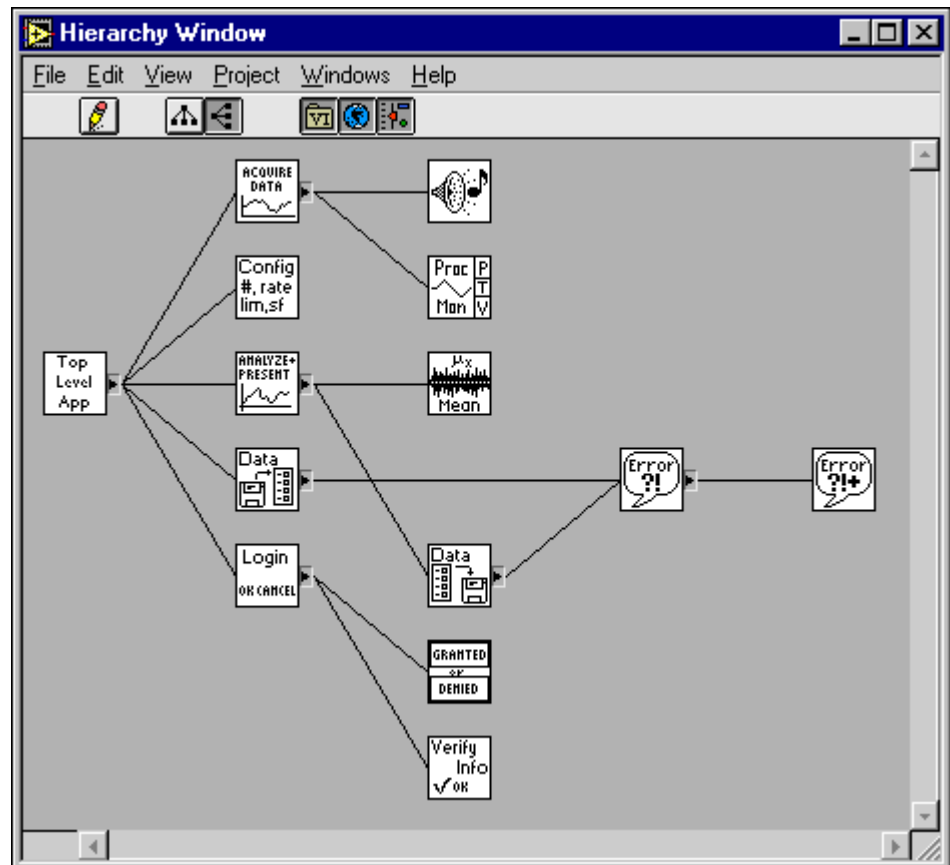
From this dialog box, you can select the VIs you want to compare, as well as the characteristics of the VIs to check. When you compare the VIs, both VIs will be displayed, along with a Differences window that lists all differences between the two VIs. In this window, you can select various differences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

Exercise 6-4

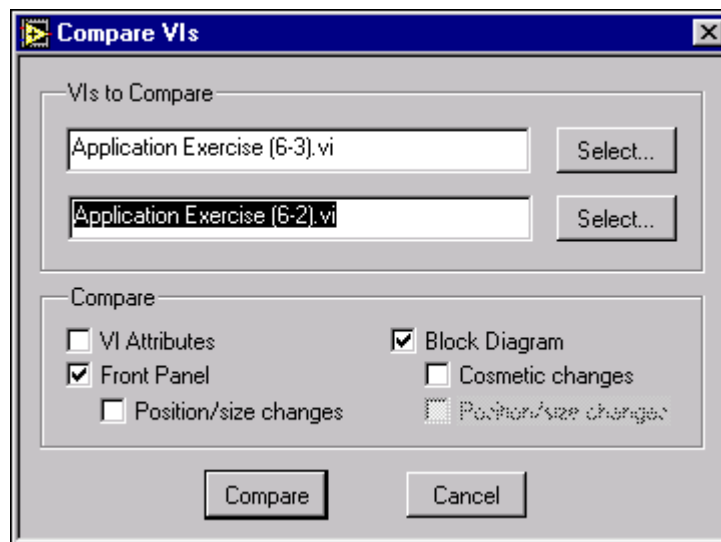
Objective: To examine some of the built-in LabVIEW features for handling applications.

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

1. Open the **Application Exercise(6-3)** VI you created in the previous exercise. Close any other VIs loaded into memory.
2. From the LabVIEW **Windows** menu, select **Show History**. The history window for the VI should appear.
3. Press **Reset...** from the History window to clear the current history. A dialog box will pop up to confirm the deletion of the history and resetting of the revision number. At this dialog box, press **Yes**.
4. In the Comment box of the History window, type in **Initial Application Created.** and then press the **Add** button. Your comment should appear in the History listing, along with a date and time stamp. Close the History window.
5. From the LabVIEW **Project** pull-down menu, select **Show VI Hierarchy**. The application's hierarchy should appear:



6. Experiment with expanding and collapsing the hierarchy. Notice that as you click on the small black arrows in the hierarchy, they expand or collapse branches of the hierarchy. You may see some icons with a red arrow by them, indicating that they call one or more subVIs. In addition, you may 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 on any subVI icon in the hierarchy to bring up the appropriate subVI. Close the subVI you selected, and close the hierarchy window.
9. Open the **Application Exercise(6-2)** VI you completed in Exercise 6-2, switch to the front panel of the **Application Exercise(6-3)** VI, and then select **Compare VIs...** from the LabVIEW project menu. The Compare VIs dialog appears:



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 above illustration.

11. Press **Compare** to bring up the Differences window and tile the two VIs. Make sure that the Circle Differences option in the Differences window is checked. Then, select a difference from the Differences listbox, select a detail from the Details listbox, and then press **Show Detail**. The difference between the two VIs will be highlighted. Examine the various differences between the two VIs and then close the Differences window.
12. Close **Application Exercise (6-2).vi**.

End of Exercise 6-4

C. LabVIEW Tools for Project Management (Optional)

There are several products you can add to the LabVIEW environment to assist in project development. Two of these products are discussed below and are included in the LabVIEW Professional Developers Suite.

Professional G Developers Toolkit

This toolkit contains several products to simplify management of larger applications. The first feature added with this toolkit 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.

In addition to SCC tools, the G Developers Toolkit contains 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.

A **File 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 File Manager can also easily 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 discussed earlier, they are used to determine the differences between two VIs or hierarchies.

Finally, a **Documentation Tool** is provided, making it easy to print documentation for the VIs in your hierarchy. With this tool, you can automate the process of printing VIs in any of the formats offered using the Print Documentation option in LabVIEW.

The manual for this toolkit not only provides reference material for the above utilities, but also extensive discussion on management of software projects. Various development models, prototyping and design techniques,

and project tracking methods are discussed in detail. Documentation of VIs and tips for developing clear G code are discussed in detail as well. Between the useful documentation and the tools added to the LabVIEW environment, the Professional G Developers Toolkit simplifies the development of larger projects.

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 program can include a hierarchy of VIs that you have created, or the program can be configured to open and run any VI available to the user.

When you purchase the LabVIEW Application Builder or the LabVIEW Professional Developers Suite, you are licensed to distribute a certain number of executable programs or run-time systems. Consult the LabVIEW Application Builder Release Notes for more information about the licensing agreement.

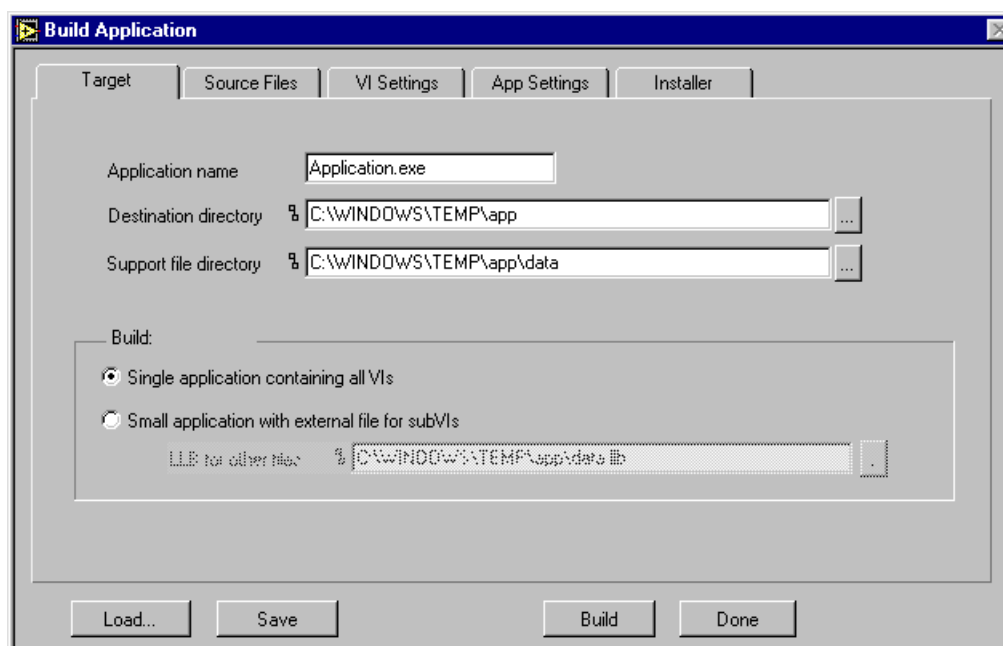
Required System Configuration

Applications 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 **Project»Build Application**, 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** dialog box contains the following tabbed pages: **Target**, **Source Files**, **VI Settings**, **App Settings**, and **Installer**.



- From the **Target** tab, you can specify 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 Setup settings. 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 panel is needed using the **VI Settings** tab.
- From the **App 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** tab (Windows only), you create an installer. The installer is written to the directory that contains your application.

When you develop an executable program with LabVIEW on Windows and ship it to another computer, you must also include the LabVIEW Run-Time DLL. The computer on which the program runs must install this DLL using the LabVIEW Run-Time DLL Installer before the program executes. If you distribute a program 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 program developed in LabVIEW. You need to include only the Run-Time DLL with the first program sent to each computer.

Exercise 6-5 Application.exe

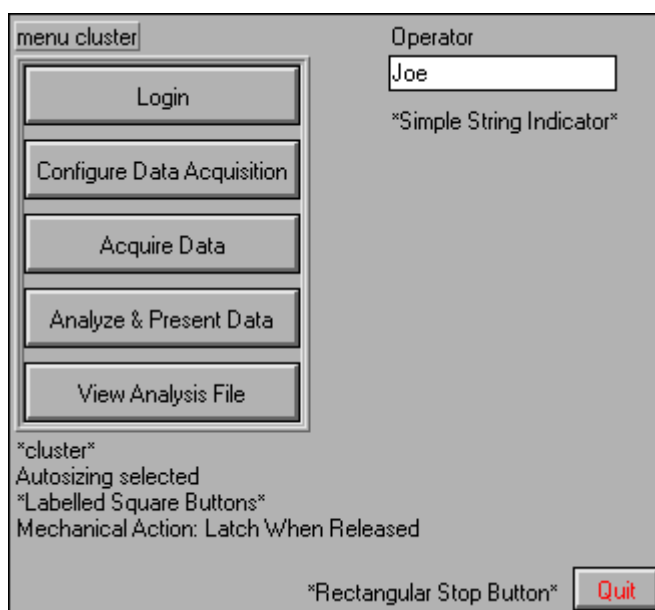
Objective: To create a stand-alone application with the Application Builder.



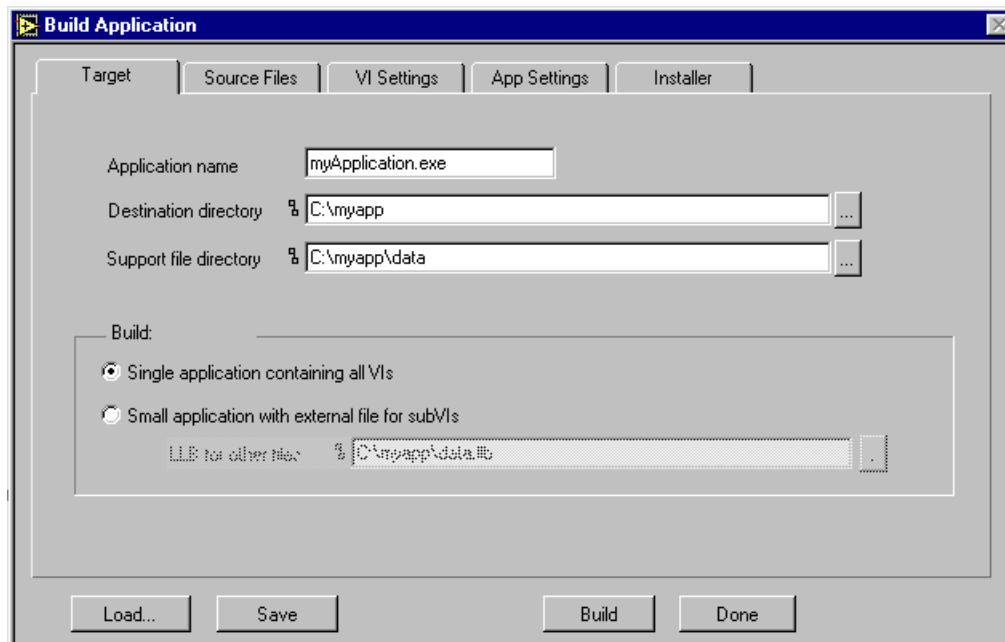
Note

You must have the Application Builder properly installed to run this example. To determine whether it is installed, pull down the Project menu in LabVIEW. If the option Build Application appears in the Project menu, then the Application Builder is properly installed.

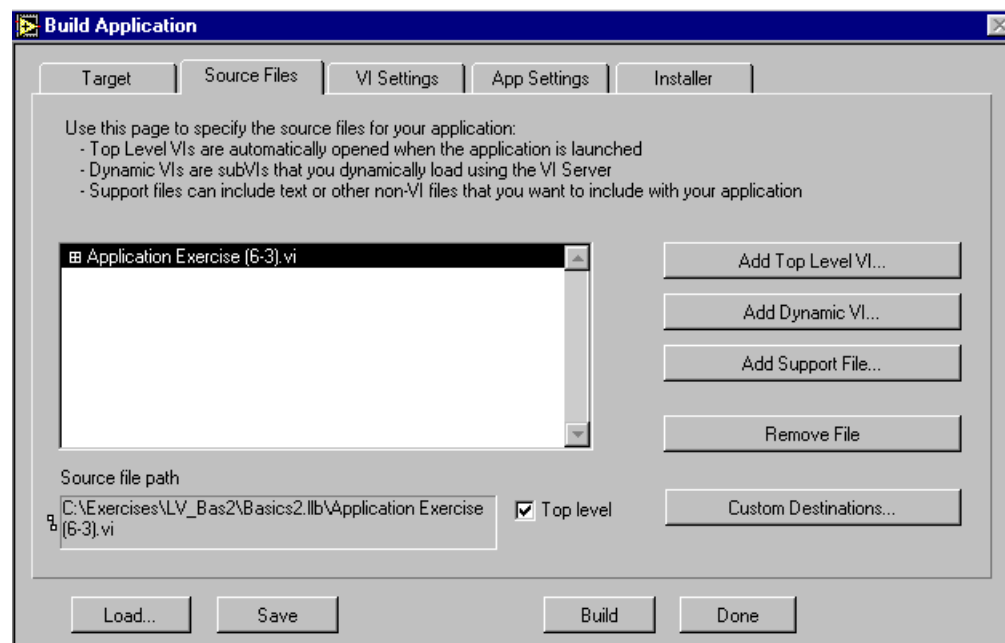
1. Open and examine the **Application Exercise(6-3)** VI you created in Exercise 6-3. Close this VI and any of its subVIs, because the Build Application utility cannot create an executable if the VIs are loaded in memory.



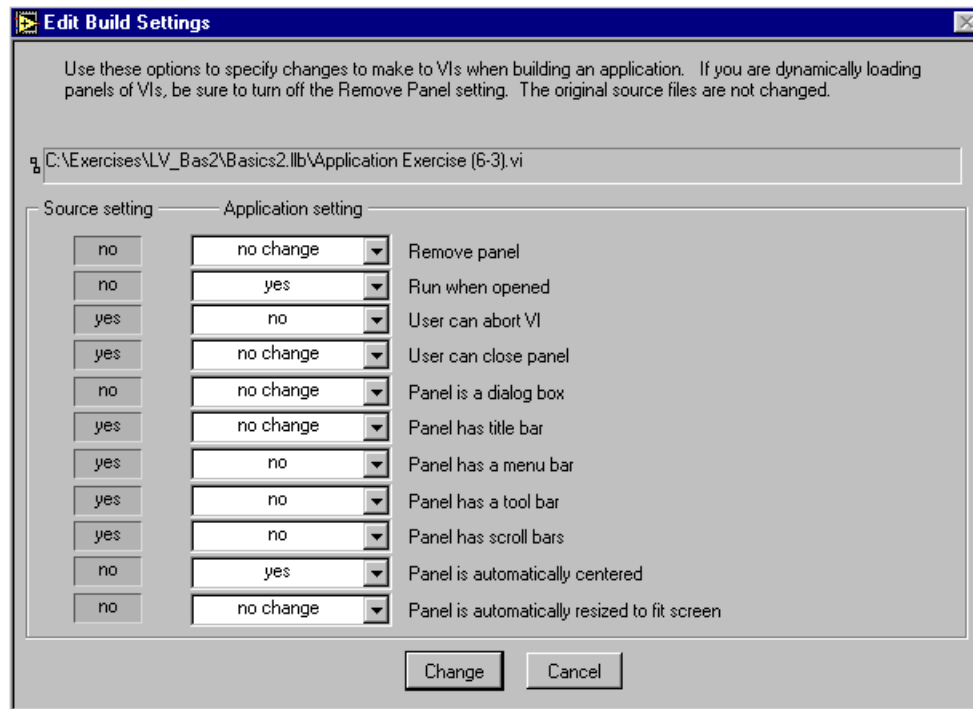
2. Select **Build Application** from the **Project** menu. You will build an executable from **Application Exercise (6-3).vi** called **myApplication.exe** and it will be placed into the **C:\myapp** directory. Modify the Target tab as shown:



3. Click on the **Source Files** tab and select the **Add Top Level VI...** button. Add **Application Exercise (6-3).vi** as shown:

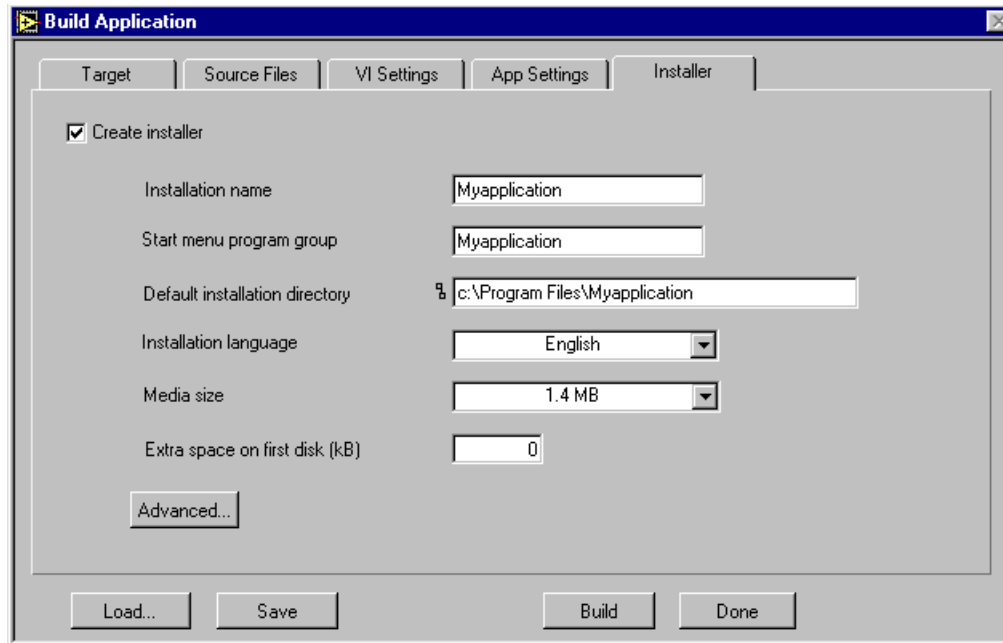


- Click on the **VI Settings** tab. You will configure the VI to run when opened, be auto-centered on the monitor, and not have scroll bars, a menu bar, or a toolbar. Highlight **Application Exercise (6-3).vi** and select the **Edit Build Settings...** button. Modify the window as shown:



- Click on the **App Settings** tab. This is where you would enable ActiveX settings or give your application a custom icon. You will leave the icon as the default LabVIEW icon. Do not change any of these settings.

- Click on the **Installer** tab. You will build a distribution kit for your application that will install into the C:\Program Files\Myapplication directory. Configure the Installer tab as shown:



- Press the **Build** button. The files associated with the installer are then compressed into setup diskettes, which will be stored in the C:\myapp\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 RunTime directory includes the LabVIEW Run-Time DLL. The executable for your application is also built and is called myApplication.exe, as defined on the **Target** tab.
- 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.
- Run myApplication.exe from the C:\myapp directory. **Application Exercise (6-3).vi** should open its panel and run automatically. Operate the VI to make sure all the settings you chose are working. Close the application when you are finished.
- Run the Setup.exe file in the C:\myapp\disks\Myapplication directory. You should be guided through a setup process, the executable will be created inside the C:\Program Files\Myapplication directory, and you should be able to run the application from **Programs>>**.

End of Exercise 6-5

Summary, Tips, and Tricks

LabVIEW has several features to assist you and your coworkers in developing your projects, such as the VI History window to record comments and modifications to a VI, and the user login, which, when used with VI History, records who made changes to a VI. You can access a VI's History window at any time by selecting **Show History** from the **Windows** 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 add-on toolkits to facilitate larger project development. The Professional G Developers Toolkit contains several utilities for managing LabVIEW code, and the Application Builder enables you to create stand-alone executables. Both of these toolkits are included in the LabVIEW Professional Developers Suite.

Additional Exercises

- 6-6 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 attribute nodes to modify the buttons.

- 6-7 Add one menu button to the menu cluster control in **Application Exercise.vi** (which you completed in Exercise 6-3) so the user can show or hide the LabVIEW Help window. The text of the button should indicate what will happen 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 Help window, you will need to add them in the **Data Operations»Description** pop-up dialog box for each control or indicator.

Make sure the VI properly tracks the state of the Help window. For example, if the user clicks the Show Help button to show the Help window, but closes the Help window from the **Show Help** option in the **Help** menu, your program will incorrectly assume that the Help window is showing. Save this VI as **Application Exercise(6-7).vi**.



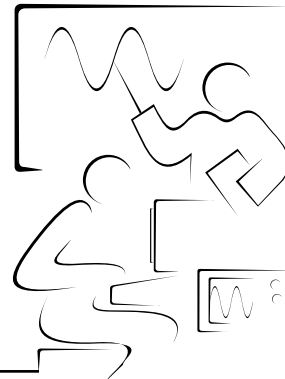
Note

*Use attribute nodes 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 Help window's visibility, and use the **Control Help Window** function to show and hide the Help window.*

Notes

Lesson 7

Performance Issues



Introduction

This lesson discusses 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 VI 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, you could execute several VIs simultaneously and the VIs still responded to user input from the mouse 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 quite 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 could 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.

LabVIEW 5, however, uses multithreading. With multithreading, different portions of a program can run under different threads (processes) in a computer system. This architecture means that the operating system can 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, you must use LabVIEW 5.0 on an operating system that supports it: Windows 95/98, Windows NT, Solaris 2, and Concurrent PowerMAX. On operating systems that do not support multithreading, LabVIEW continues to operate with cooperative multitasking.

Benefits

One important benefit of multithreaded LabVIEW is the separation of the user interface from 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 on 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

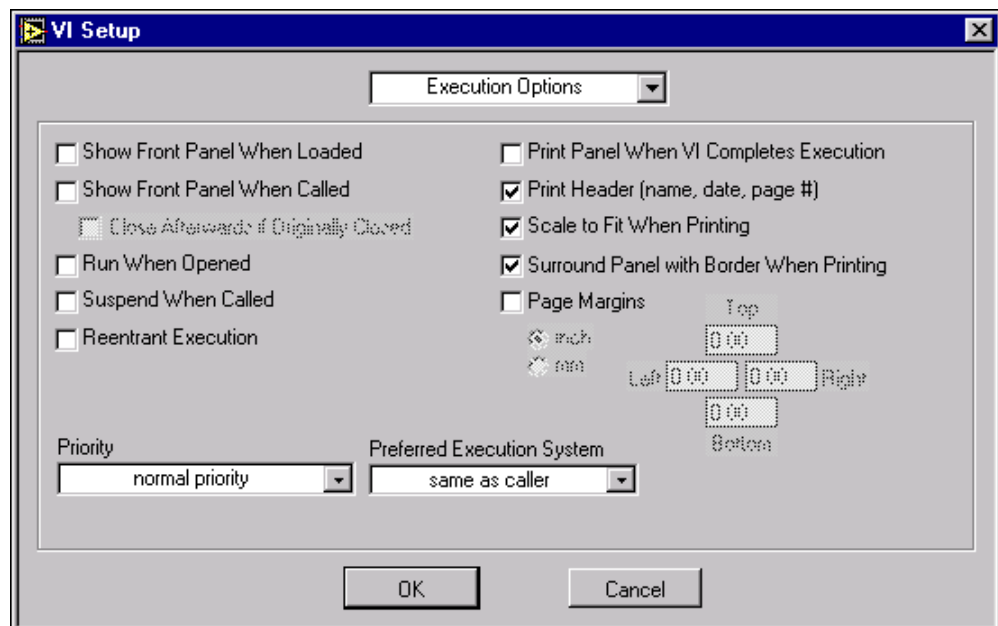
When an existing LabVIEW application runs, it takes advantage of the multithreaded system automatically without any modification to the

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 may want to move appropriate VIs to different execution systems. If you are developing instrument drivers, for example, 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, pop up on the icon pane in the VI's front panel or block diagram, select **VI Setup**, then select **Execution Options** from the drop-down menu. The following dialog box appears:



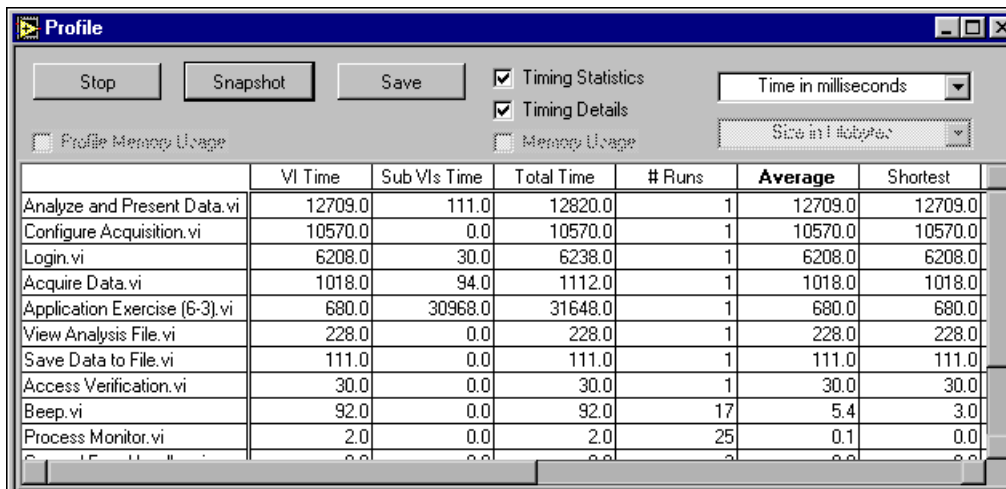
The **Preferred Execution System** list box lists the available categories of execution systems. You can also prioritize parallel tasks in a multithreaded environment by setting the **Priority** of the VI. Within each thread category, you can specify the priority of execution: normal, above normal, high, 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. VIs with a subroutine priority level, however, will behave differently than VIs with other priority levels. When a VI runs at subroutine priority, it executes in the thread category of its caller, and no other VI can execute 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 execute a simple computation with no interactive elements. You can skip the execution of a subroutine priority subVI when it is busy by popping up on 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.

You must 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 can also be a very effective way of optimizing your LabVIEW code. For more information on how to use Wait functions, refer to the Understanding How LabVIEW Executes VIs topic in the LabVIEW online help, and the Using Wait Functions to Prioritize Tasks section in Chapter 26, *Understanding the G Execution System*, of the *G Programming Reference Manual*.

B. The VI Profile Window

The VI 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 can also monitor subVI performance within different VIs. To show the Profile window, select **Show Profile Window** from the **Project** menu. The window will appear similar to the window shown below.



The screenshot shows the 'Profile' window with a toolbar containing 'Stop', 'Snapshot', and 'Save' buttons. Below the toolbar are checkboxes for 'Timing Statistics' (checked), 'Timing Details' (checked), 'Profile Memory Usage' (unchecked), and 'Memory Usage' (unchecked). To the right of these checkboxes are dropdown menus for 'Time in milliseconds' and 'Sort by'. The main area of the window contains a table with the following data:

	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 choose 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, you simply click on 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, and also 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. Consult the **Online Reference** or the *LabVIEW User Manual* for more information about these statistics.

The SubVIs statistic 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 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 execute. 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, you must select the **Profile Memory Usage** option before starting the profiling session. You can also select the **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 just VI execution.

The Average Bytes statistic 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.

C. Speeding Up Your VIs

LabVIEW compiles your VIs and produces code that generally executes very quickly. But when working on time-critical applications, you need to use programming techniques to obtain the best possible execution speed. There are three areas to consider 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)

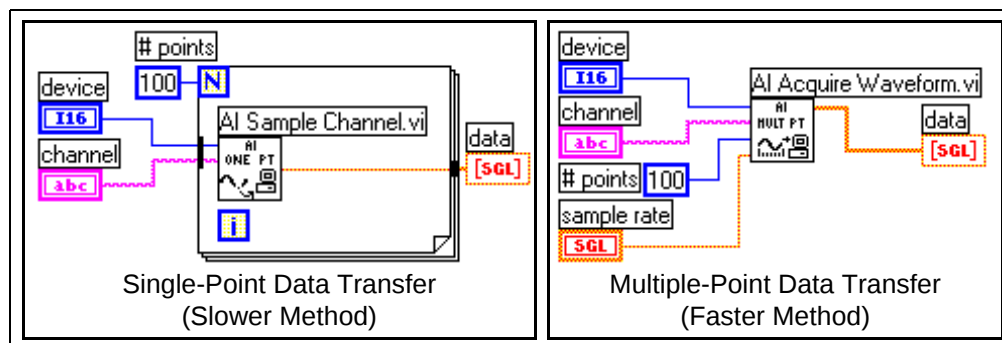
Other factors, such as execution overhead and subVI call overhead, usually have minimal effects on execution speed.

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, you should 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, you could use the **AI Sample Channel VI** in a loop with a wait function to establish the timing. But the faster method is to use the **AI Acquire Waveform VI** with an input specifying that you want 100 points.



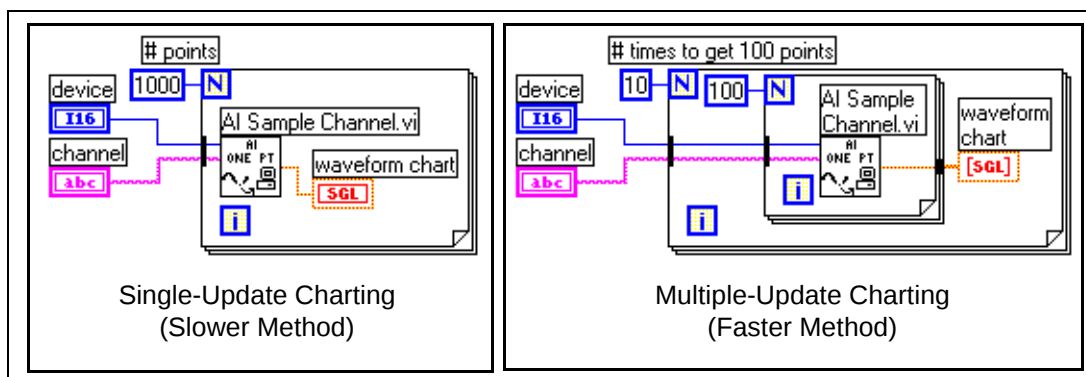
In the previous example, overhead for the **AI Acquire Waveform** VI is roughly the same as the overhead for a single call to the **AI Sample Channel** VI, even though it is transferring much more data. In addition, the data collected by the **AI Acquire Waveform** VI uses hardware timers to control 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 very 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.

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

As shown below, you can reduce display overhead by minimizing the number of times your program updates the display. Drawing data on the screen is an I/O operation, just as 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

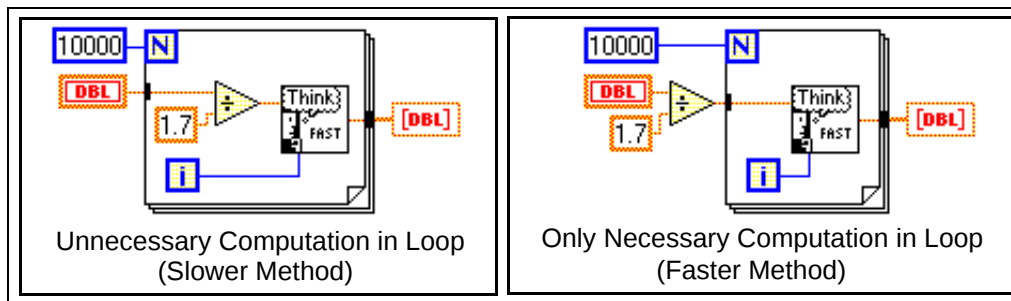
SubVI Overhead

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 may consider embedding the loop in the subVI.

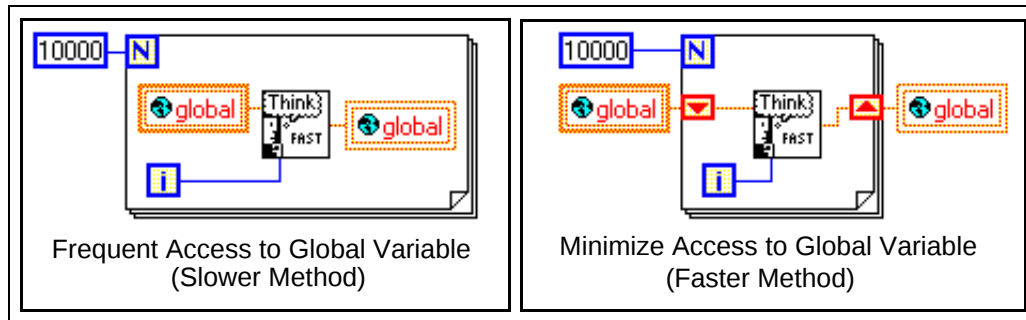
Another way to minimize subVI overhead is to turn your subVIs into subroutines (using the VI Setup priority option). However, there are some trade-offs. Subroutines cannot display front panel data, call timing or dialog box functions, or multitask with other VIs. Subroutines are generally most appropriate for VIs that do not require user interaction and are short, frequently executed tasks.

Unnecessary Computation in Loops

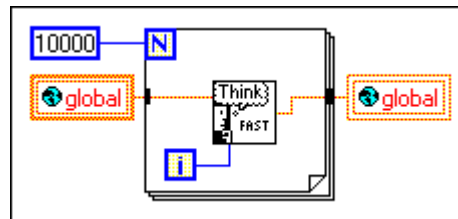
Avoid placing calculations in loops if the calculation produces the same value for every iteration. Instead, move the calculation out of the loop and pass the result into the loop. For example, consider the following two diagrams. The result of the division is the same every time through the loop; therefore,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



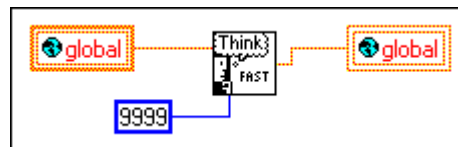
Also, avoid accessing local and global variables unnecessarily in loops. For example, the first diagram shown below wastes time by reading from the global and writing to the global during each iteration of the loop. If you know that the global variable is not read from or written to by another diagram during the loop, consider using shift registers to store the data as shown in the second 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. But without using a shift register, the results from the subVI are never passed back to the subVI as the new input value, as shown below.

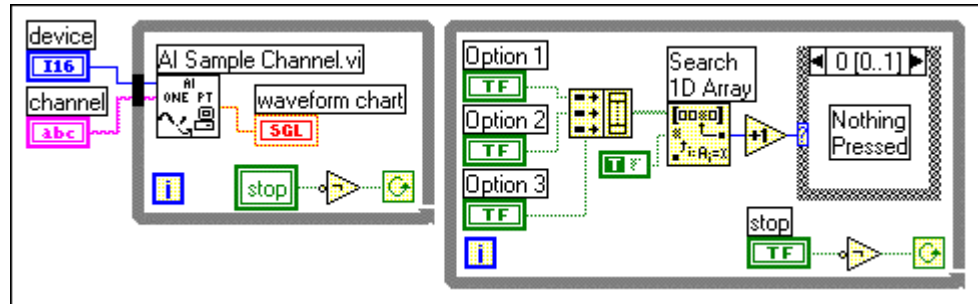


In the diagram above, the global variable is read once before the loop begins, and the same value is passed to the function 10,000 times. The result of the loop is the same as if you had written the code shown below.

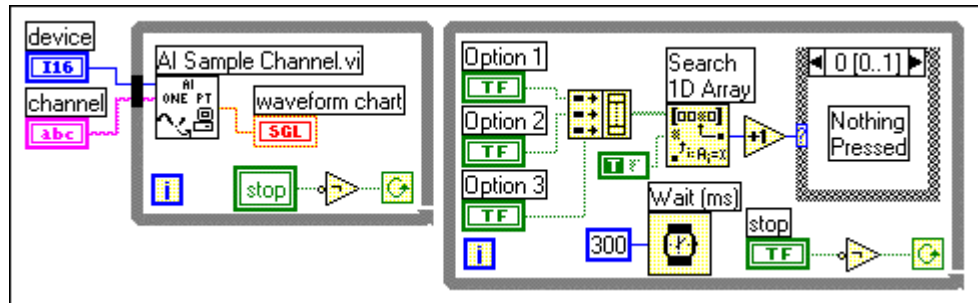


Parallel Diagrams

When you have several loops running in parallel on a diagram, LabVIEW switches 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 diagram.



The two loops run in parallel. One of the loops is acquiring data and must execute as frequently as possible. The other loop is monitoring 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 below, you can call the **Wait (ms)** function in the user interface loop to give significantly more execution time to the other loop.

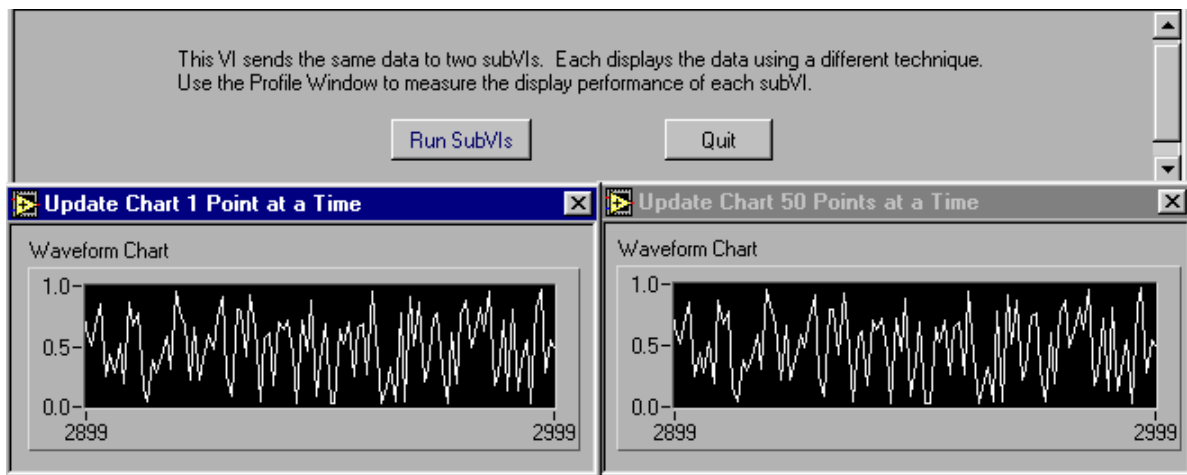


Exercise 7-1 Charting Benchmark Example.vi

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

When you open the **Charting Benchmark Example VI**, two other subVI panels will 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 98/95/NT, Solaris2, or Concurrent PowerMAX), follow steps a through c below. If you are on a system that does not support multithreading, skip to step 2.
 - a. You will first configure LabVIEW so that it does not use multithreading when it runs VIs. From the LabVIEW **Edit** menu, select **Preferences....** The Preferences dialog box appears.
 - b. Change the top pull-down menu of the Preferences dialog box to **Performance and Disk**. In the Performance and Disk section, uncheck the **Run with multiple threads** dialog box. 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 may have open.
3. Open the **Charting Benchmark Example VI** in Basics2.11b. The VI is already built for you. When you open it, two other VIs will 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.

4. Select **Show Profile Window** from the **Project** menu. Move the window so that it does not overlap any of the VI front panels.
5. Click on the Start button in the Profile window. You will 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 press the Run SubVIs button. Run the subVIs at least three or four times. Then press the Quit button to stop the **Charting Benchmark Example VI**.
7. After running the subVIs several times, click on the Snapshot button in the Profile window. Locate **Charting Benchmark Example.vi** in the Profile window and double-click on 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**.

For more detailed information about these statistics, consult the **Online Reference** or the *LabVIEW User Manual*.

9. Click on 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, check the **Run with multiple threads** option in the LabVIEW preferences and restart LabVIEW. Repeat steps 3 through 10, noting how much faster the VIs execute when the display is running under its own execution subsystem. You may want to change the Profile window to display **Time in milliseconds** to make it easier to read.

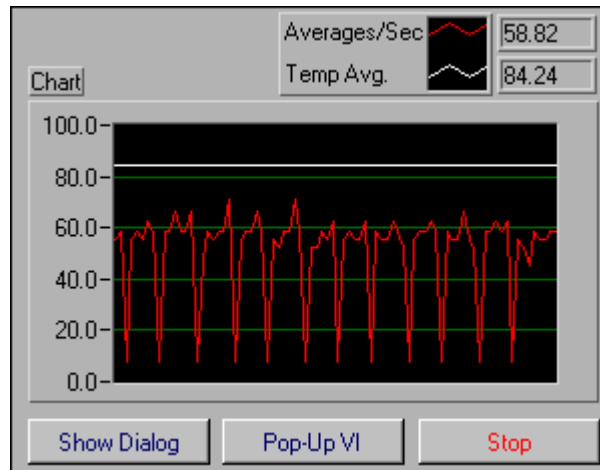
End of Exercise 7-1

Exercise 7-2 Dialog & SubVI Demo.vi

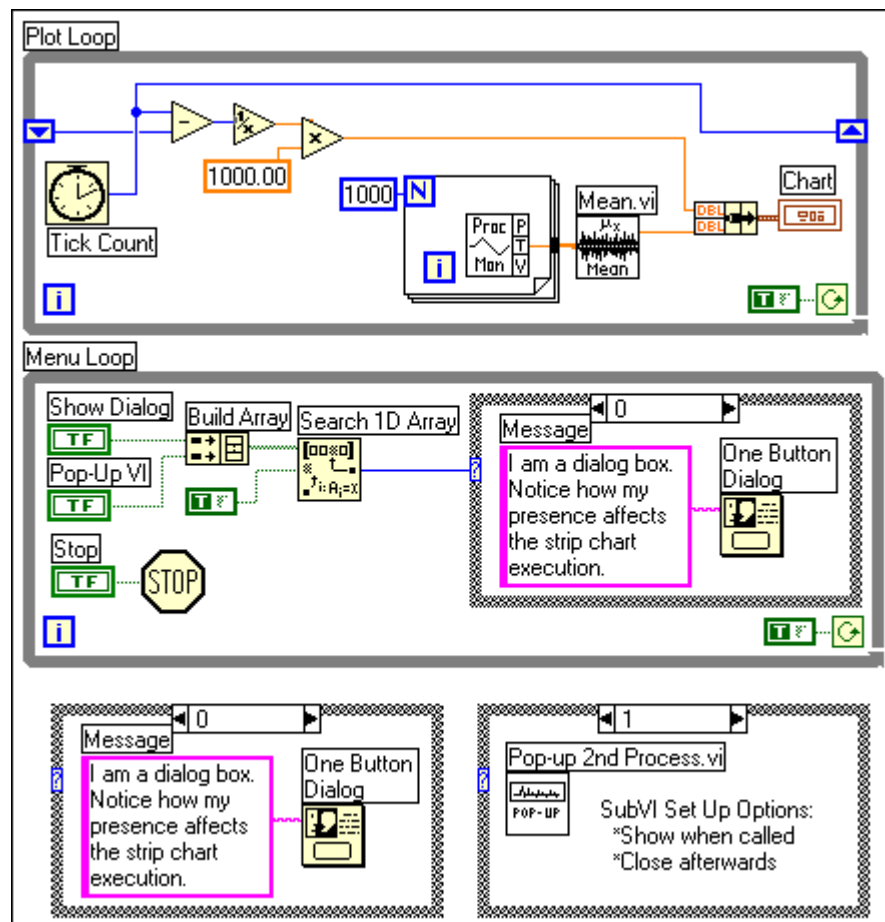
Objective: To observe how dataflow structure can affect execution speed.

You will load and run a VI that plots a waveform pattern. You should especially note how the dataflow affects the execution of this program.

Front Panel



1. If you are running LabVIEW on an operating system that supports multithreading (Windows 98/95/NT, Solaris2, or Concurrent PowerMAX), follow steps a through c below. If you are on a system that does not support multithreading, skip to step 2.
 - a. You will first configure LabVIEW so that it does not use multithreading when it runs VIs. From the LabVIEW **Edit** menu, select **Preferences....** The Preferences dialog box appears.
 - b. Change the top pull-down menu of the Preferences dialog box to **Performance and Disk**. In the Performance and Disk section, uncheck the **Run with multiple threads** dialog box. 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 in Basics2.11b. The VI is already built for you. The front panel contains a strip chart and several option buttons.



- Open the block diagram and examine it. Two While Loops will run in parallel.
- Run the VI. The chart shows the number of averages per second and an average temperature pattern. It updates point by point. Notice the plot speed and the averages/second being calculated.

Notice that the Averages/Sec value periodically 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 is executing and checking the values of the buttons on the panel. Performance increases when the Plot Loop is running.

- Press 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 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 press the Show Dialog button, Case 0 is executed. The case does not complete until you press 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 on the **Pop-Up VI** button. The **Pop-up 2nd Process** VI will open its panel and run. Notice the effect on the first VI's Averages/Sec value. As in the situation when you pressed the Show Dialog button, the Menu Loop is no longer executing 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 is running. Therefore, LabVIEW executes multiple VIs in the same way it executes parallel loops—each process gets an equal time-slice.
7. Click on 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, check the Run with Multiple Threads option in the LabVIEW preferences. Exit and restart LabVIEW. Repeat steps 2-9 and notice how multithreading affects the execution speed of the **Dialog & SubVI Demo** VI.

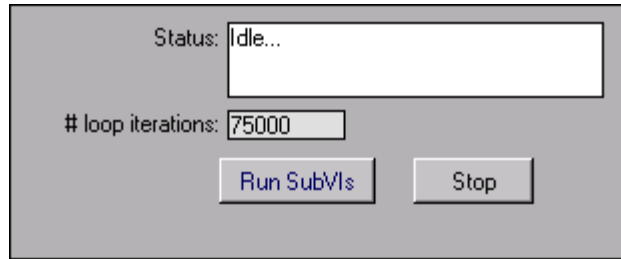
End of Exercise 7-2

Exercise 7-3 Computation & Global Benchmark.vi

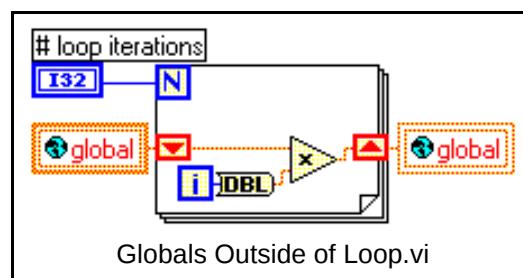
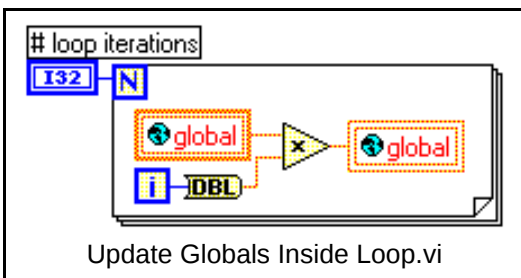
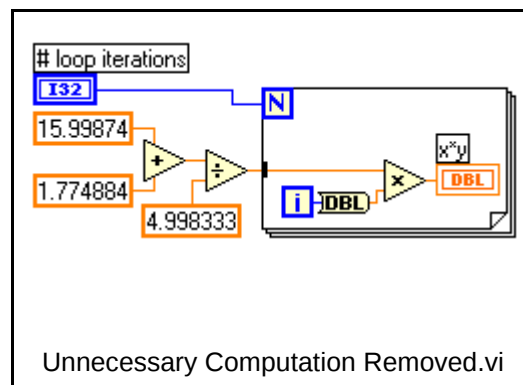
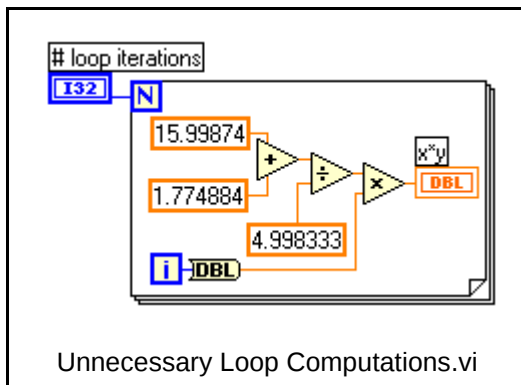
Objective: To observe the effects of unnecessary computation and global variable activity on VI execution speed.

You will use the VI 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 may have open.
2. Open the **Computation & Global Benchmark** VI in Basics2.11b. The VI is already built for you.
3. Select **Show Profile Window** from the **Project** menu. Click on the Start button in the Profile window. Do not gather any memory statistics.
4. Run the **Computation & Global Benchmark** VI. Click on the Run SubVIs button to run the four subVIs. Run the subVIs at least three or four times. The block diagrams for these four VIs appear below.



- After running the subVIs several times, stop the VI and click on the Snapshot button in the Profile window. Locate the **Computation and Global Benchmark.vi** in the Profile window and double-click on 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?

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

End of Exercise 7-3

D. System Memory Issues

LabVIEW transparently handles many of the details that you normally deal with in a conventional, text-based language. One of the main challenges of programming with a conventional language is memory use. In a conventional 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 conventional 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 diagram with graphical connections representing the transition of data.

Functions that generate data allocate storage for that data. When data is 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 programs 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 the 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 98/95/NT, 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 may occasionally notice more sluggish performance because data is 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.

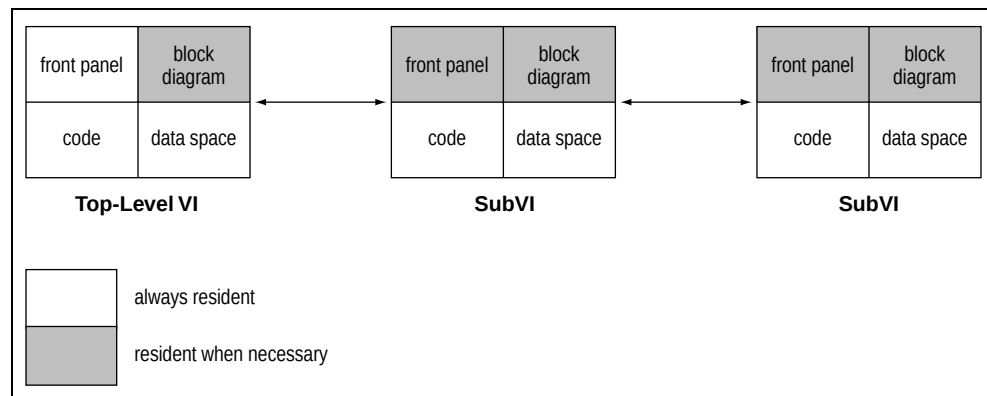
VI Components

VI's have the following major components:

- Front Panel
- Block Diagram
- Code (diagram compiled to machine code)
- Data (control and indicator values, default data, 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 loads the diagram into memory as well.

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



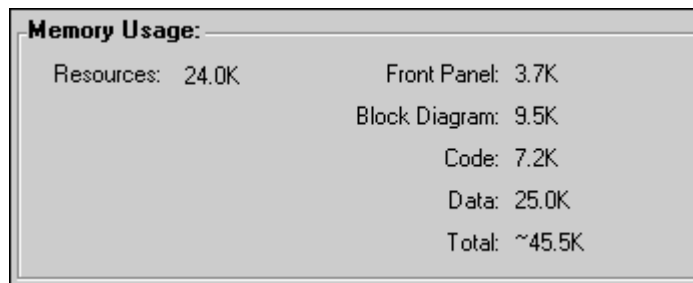
As shown above, you can save memory by converting some of your VI components into subVIs. If you create a single, large VI with no subVIs, you end up with everything (the front panel, code, and data) for that top-level VI in memory. If the VI is broken into subVIs, the code for the top-level VI is smaller,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

E. Optimizing VI Memory Use

This section discusses 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 illustration, you can use the **Show VI Info** option from the **Windows** menu to get a breakdown of the memory use for a specific VI. The left column summarizes disk use and the right column summarizes how much RAM is being used for the various components of the VI. Notice that the information does not include memory use of subVIs.



You can also use the VI 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 memory information being reported.

General Rules for Better Memory Use

The following rules should help you create VIs that use memory efficiently.

- Breaking a VI into subVIs will usually improve your 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 panels, display large arrays and strings only when necessary. Indicators on open panels retain a copy of the data that they display.
- If the panel of a subVI will not be displayed, do not leave unused attribute nodes on the subVI. Attribute nodes cause the 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 panel for the subVI needs to be loaded for range checking, and extra copies of data are made for the subVI controls and indicators.
- In designing 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** functions, 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. Whenever 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 Data vs. Complicated Data Structures* section later in this lesson contains more information about designing efficient data types.

LabVIEW Data Types

Understanding how LabVIEW stores data in memory will help you create more efficient I/O applications. Each wire and terminal in the LabVIEW block diagram has a data type associated with it. You are already familiar with the basic data types, such as integers, floating-point numbers, and strings. You have also used arrays of these data types. This lesson will further discuss LabVIEW data structures and some ways to manipulate the data.

Numeric Data Types

While working with numeric data in LabVIEW, you have probably noticed that you can change the *representation* of numeric constants, controls, and indicators. The representation indicates how LabVIEW stores the object's data in memory. Different numeric representations use a different number of bytes of memory to store data. They also assign data as *signed* (with both negative and positive values) or *unsigned* (with only zero or positive values).

The following table lists the numeric data types available in LabVIEW, along with their abbreviation and a sample digital control.

Data Type	Abbreviation	Control
Signed 8-bit integer (Byte)	I8	
Signed 16-bit integer (Word)	I16	
Signed 32-bit integer (Long)	I32	
Unsigned 8-bit integer	U8	
Unsigned 16-bit integer	U16	
Unsigned 32-bit integer	U32	
Single-precision floating-point number (32-bit)	SGL	
Double-precision floating-point number (64-bit)	DBL	
Extended-precision floating-point number (Windows: 80-bit; Macintosh: 96-bit; Sun, HP-UX: 128-bit)	EXT	
Complex single-precision floating-point number (64-bit)	CSG	
Complex double-precision floating-point number (128-bit)	CDB	

Data Type	Abbreviation	Control
Complex extended-precision floating-point number (Windows: 160-bit; Macintosh: 192-bit; Sun, HP-UX: 256-bit)	CXT	

Arrays of Numeric Data

When you create an array of numeric data in LabVIEW, the data is stored in a *contiguous* block of memory. This means that the elements in the array are stored in adjacent memory locations. In addition to storing the data for a one-dimensional array, LabVIEW stores the number of elements in the array in a Long integer. Therefore, the largest array you can create would contain $2^{31} - 1$ elements. The following table shows how LabVIEW stores a one-dimensional array of 16-bit integers in memory.

N	e_0	e_1	e_i	e_j	e_{N-1}
no. elements	element 0	element 1	element i	element j	element N-1
4 bytes	2 bytes	2 bytes	2 bytes	2 bytes	2 bytes

When you create arrays containing more than one dimension, LabVIEW stores the number of elements in each dimension in an array of Long integers, followed by the data. Each dimension in the array can contain up to $2^{31} - 1$ elements.

The following illustrations show how LabVIEW stores a two-dimensional array of 16-bit integers, with three rows and two columns.

Original Data

5	3
76	33
12	18

How LabVIEW Stores the Two-Dimensional Array in Memory

3	2	5	3	76	33	12	18
No. rows	No. columns	r0, c0	r0, c1	r1, c0	r1, c1	r2, c0	r2, c1
4 bytes	4 bytes	2 bytes	2 bytes	2 bytes	2 bytes	2 bytes	2 bytes

LabVIEW Strings

LabVIEW stores a string in memory the same way it stores an array of unsigned byte integers. The length of the string—as a four-byte, signed integer—precedes the first character in the string. Thus, the longest string you can create would contain $2^{31} - 1$ characters, or 2 GB of information. The following table shows how LabVIEW stores a string in memory.

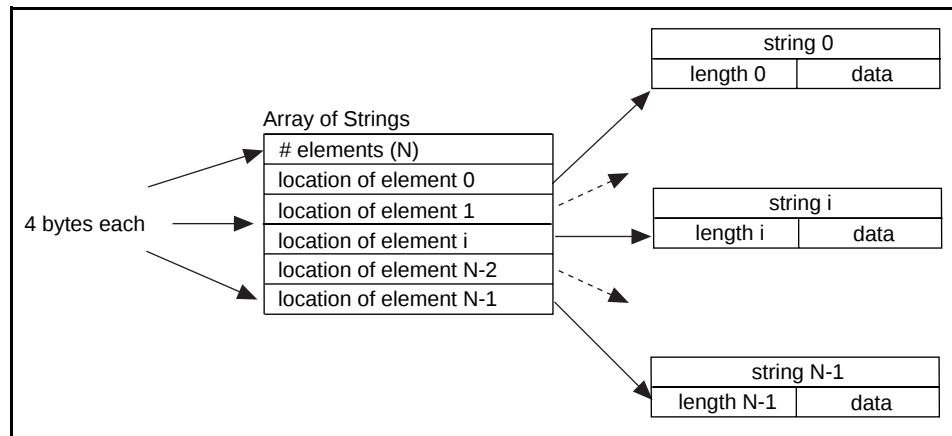
N	abc...
No. of characters (4 bytes)	string data

You can use a LabVIEW string to store virtually *any* kind of data. The string data type can store normal text as well as binary data. Therefore, strings are very useful in serial, GPIB, VXI, TCP (Transmission Control Protocol), and file I/O applications. For example, you use the string data type whenever you use LabVIEW to read information from a GPIB, serial, or VXI instrument, or to send or receive data over a network using TCP. These methods of data transfer frequently send binary information, such as the data points in a sampled waveform.

Because a LabVIEW string can contain either textual or binary information, string controls and indicators have several different viewing modes. Two of these modes are useful for viewing binary information. The Hex Display mode displays the binary data directly as bytes of information. The ‘\’ Codes Display mode shows text characters normally, but displays spaces, tabs, and carriage returns as \s, \t, and \r, respectively. This display mode is useful when you want to verify string formatting, because it is difficult to identify “white space” characters in a normal text display. Non-viewable characters are shown as a backslash character followed by two digits in hexadecimal notation.

Arrays of Strings

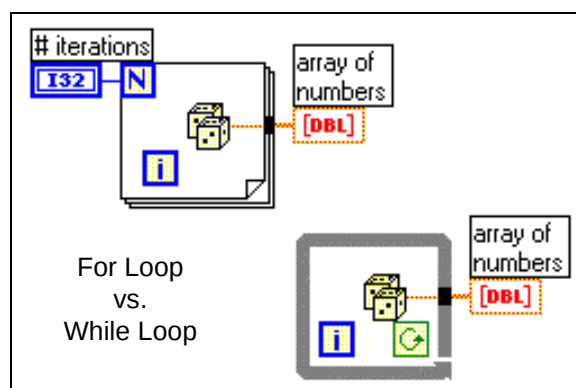
An array of strings is a more complex data type than an array of numbers, because each element in an array of strings can contain a different number of characters. (In contrast, each element in an array of numeric data uses the same number of bytes of memory to store its data.) Therefore, LabVIEW stores arrays of strings differently—in *tree* form. That is, the data is stored in separate, discontinuous locations. When you work with an array of strings in LabVIEW, an array stores the locations of the strings in memory. To access an element in an array of strings, LabVIEW finds the element in this array of memory locations to find the string and then accesses the string. The following illustration shows how LabVIEW stores an array of strings in memory.



Assembling and Processing Arrays

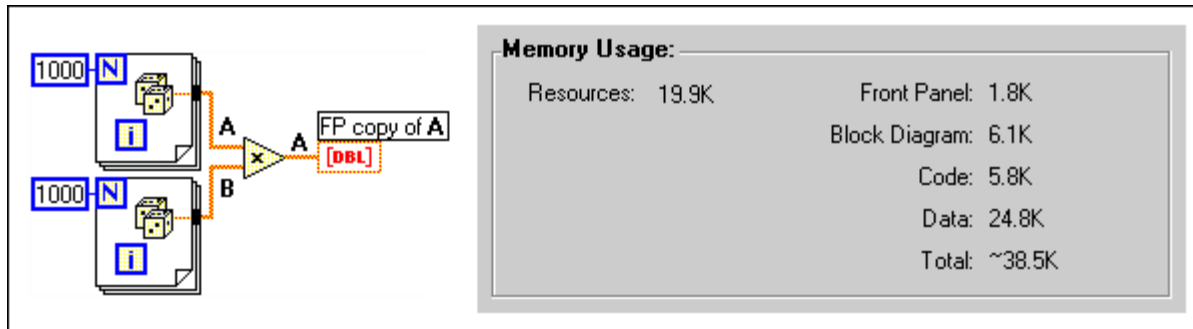
When designing your block diagram, there are several steps you can take to make your VI use memory more efficiently. For example, you can assemble and process arrays in ways that minimize the amount of memory access required.

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 may 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 whenever possible, rather than using While Loops or concatenating arrays with the **Build Array** function.

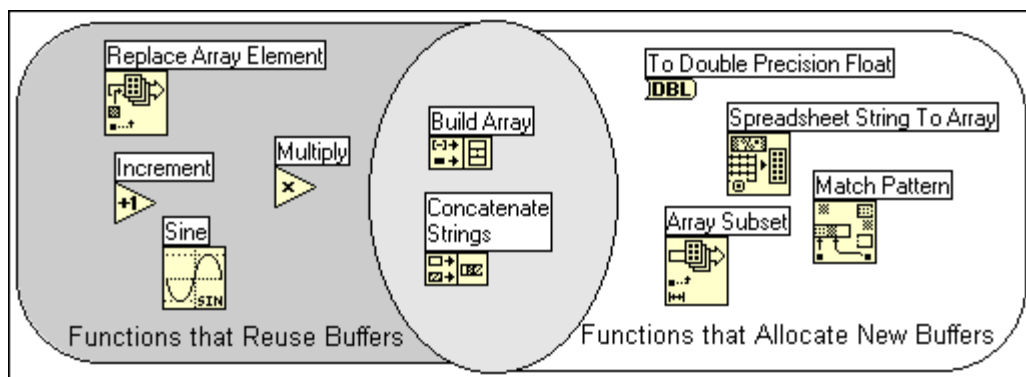


Inplaceness

Whenever possible, LabVIEW's compiler reuses a function's input buffers to store its output. This buffer sharing is called *inplaceness*. In the example below, the **Multiply** function output uses the same buffer as the top input. The array at the right is said to be inplaceness 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. In the following illustration, A, B, C, and D identify individual buffers. **Build Array** and **Concatenate Strings** are special functions. They operate inplaceness whenever they can, but sometimes they must allocate new buffers.

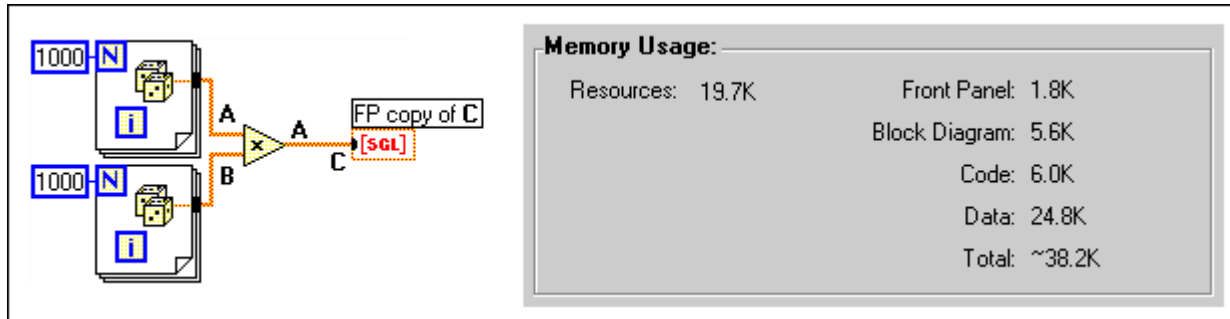


Coercion and Consistent Data Types

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.

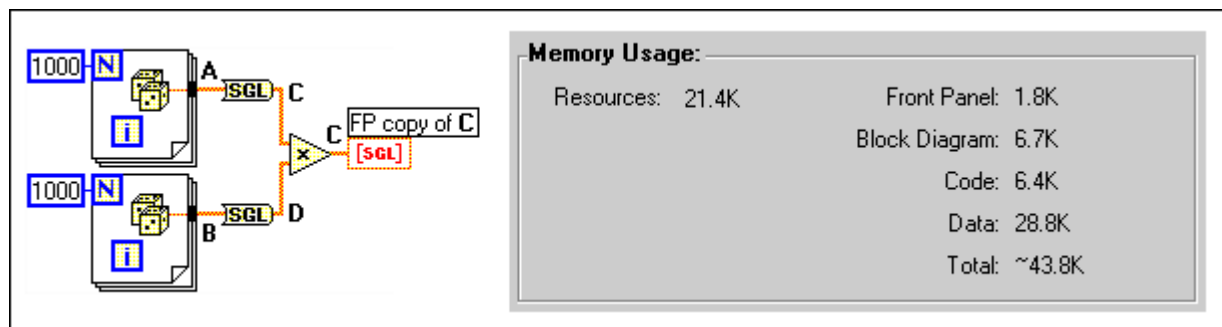
(Recall that each DBL value requires 8 bytes of memory, while each SGL value requires 4 bytes of memory.)

Method 1 (Incorrect)



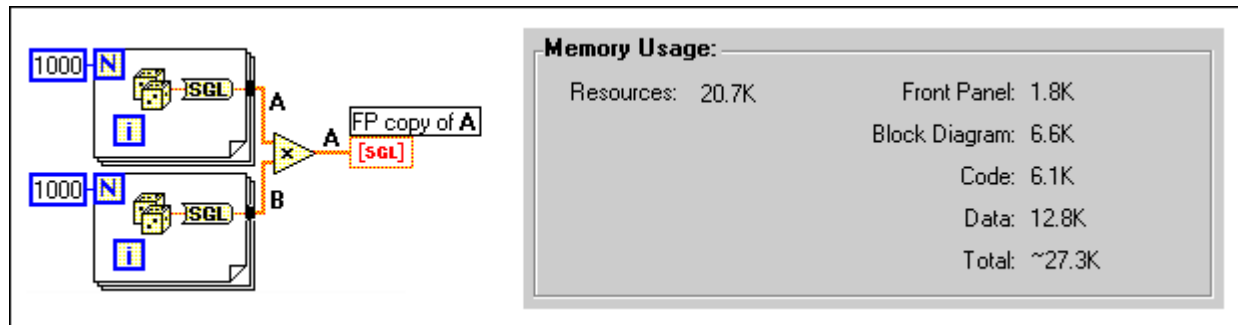
Method 1 might be your first attempt to save data space memory. You may think changing the representation of the array on the front panel (FP) to SGL might save space memory. However, this method does not affect the amount of memory needed by the VI, because the function creates a separate buffer to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

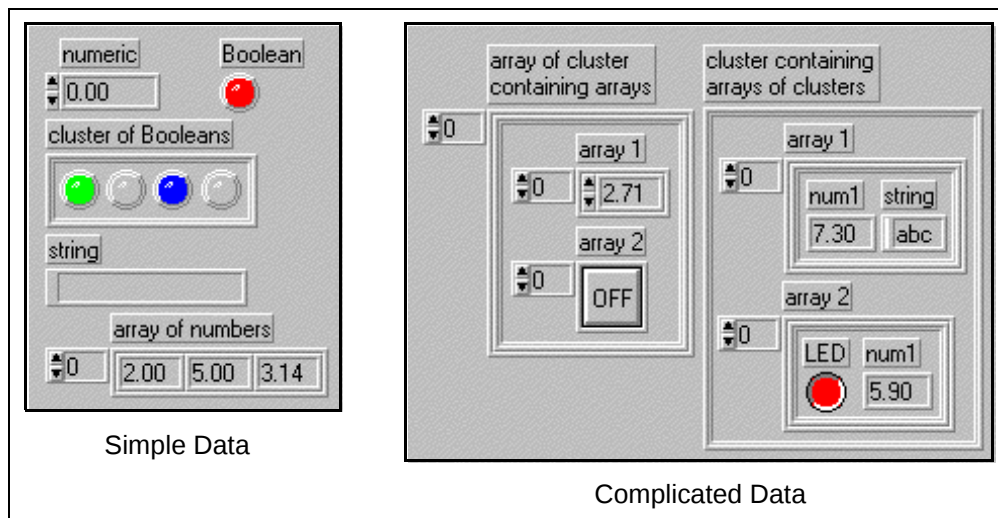
Method 3 (Correct)



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.

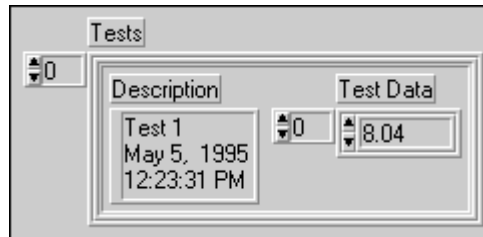
Simple vs. Complicated Data Structures

Simple data types, which include strings, numbers, Booleans, clusters of numbers or Booleans, and arrays of numbers or Booleans, are easily 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.

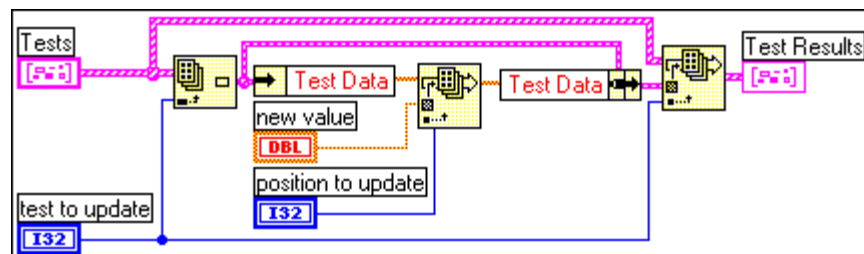


For the best performance, avoid creating complicated data structures. Performance can suffer because it is difficult to access and manipulate the interior elements without generating copies of data. Therefore, keep your data structures as “flat” as possible. Flat data structures can generally be manipulated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

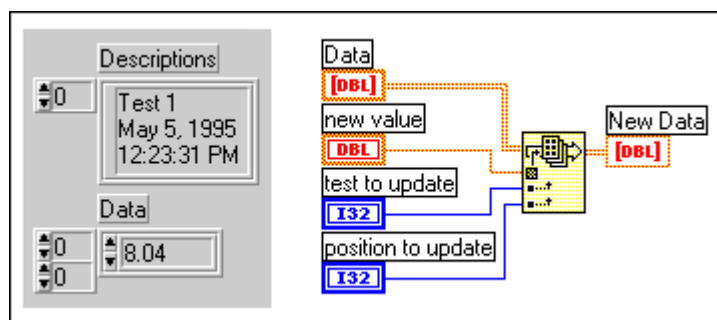
Consider an application in which you want to record the results of several tests. In the results, you want a string describing the test and an array of test results. One data type that you might use to store this information is an array of clusters containing a description string and an array of test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



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 below. 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 example shown below, the front panel contains new controls to store the data and the diagram performs the same change to the test data as shown above.

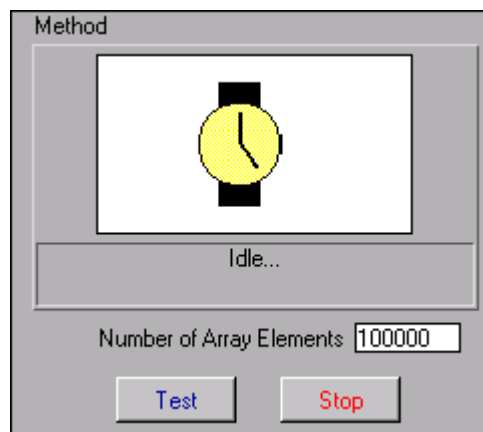


Exercise 7-4 Assembling Arrays.vi

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

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

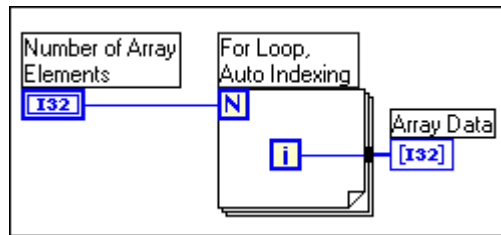
Front Panel



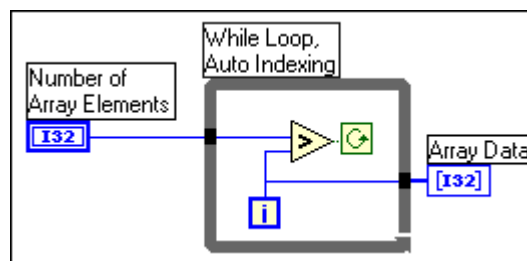
1. Close all other VIs that you may have open.
2. Open the **Assembling Arrays** VI in Basics2.11b.
3. Select **Show Profile Window** from the **Project** menu. Click on the **Start** button in the Profile window to begin a profiling session. Do not gather any memory statistics.
4. Run the **Assembling Arrays** VI. To test the different array-building methods, press 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 on the **Snapshot** button in the Profile window. Double-click on the **Assembling Arrays.vi** to show its list of subVIs if they are not already showing.

A brief discussion of the six different array-building techniques tested by this VI appears on the following page.

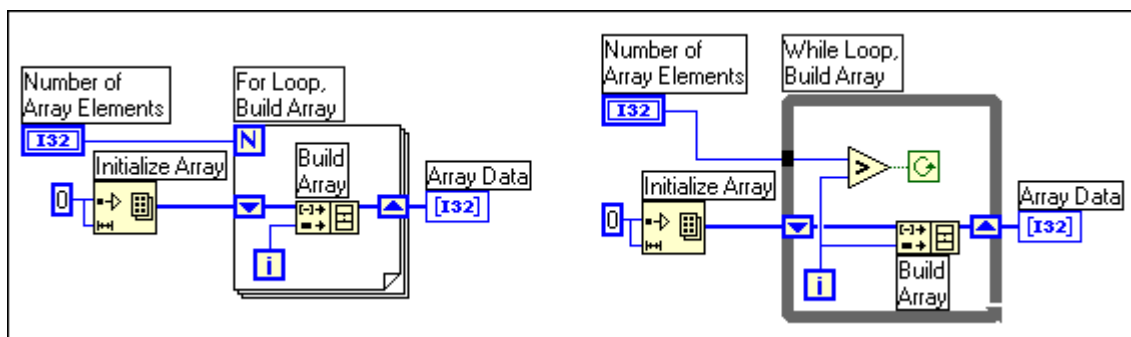
If you use a For Loop to assemble an array, LabVIEW can allocate the necessary space prior to the first iteration of the loop. For example, the array uses $N \cdot 4$ bytes of memory. (The iteration terminal generates a Long integer (I32) = 4 bytes.)



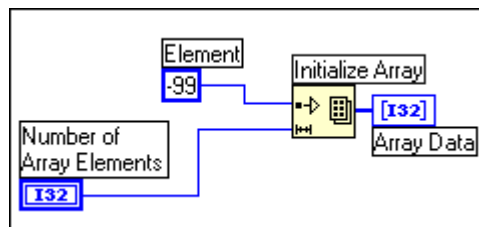
Because the initial array size is unknown, auto-indexing in a While Loop is not quite as efficient as auto-indexing in a For Loop. With auto-indexing enabled in a While Loop, LabVIEW resizes the buffer containing the array in large increments as necessary.



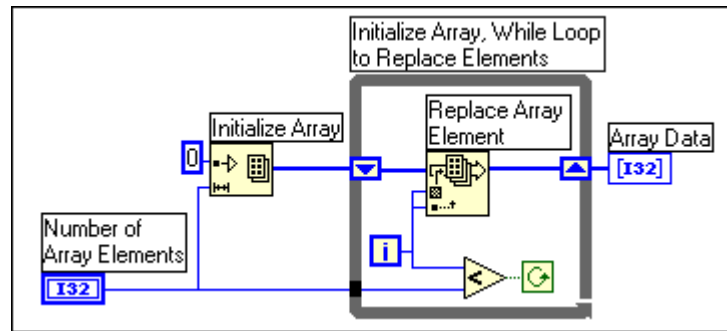
Avoid using the **Build Array** function inside a loop. Every time a new value is appended to the array, LabVIEW must reallocate the buffer, which may also require relocating the entire array in memory.



Although the **Initialize Array** function is not necessarily faster than the For Loop, it is very flexible and can be helpful when manipulating arrays.



If you cannot predetermine the number of elements a loop will generate, but you can place an upper limit on the value, generate an array containing the maximum number of elements the array will need. Then, use a While Loop and the **Replace Array Element** function to place values into the array as they are generated.



6. Stop and close the Profile window. After you have finished testing the VI, close it. Do not save changes.

End of Exercise 7-4

Exercise 7-5 Performance Challenge.vi/ Maximized Performance.vi

Objective: To improve the performance of a VI.

A VI was created to perform the following tasks:

1. Generate four runs of data. Each run should consist of an array of 5000 data points and a string containing a time stamp of when the data was taken. Each element needs only digits of precision.
2. Save 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. Find 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.

The VI that was created for this task is poorly written, executing very slowly and taking much more memory than necessary. In this exercise, you will optimize this VI to increase its execution speed while decreasing its memory requirements.

1. Close all VIs you may have open.
2. Open the **Performance Challenge** VI in Basics2.11b. Show the Profile window by selecting **Project»Show Profile Window**. 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 execution time.
3. Run the VI. After it finishes, click on 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. From the LabVIEW **Windows** menu, select **Show VI Info....** Notice the considerable amount of memory taken by the data in this VI.
5. Modify the VI to improve the execution 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 execution 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.

Hint: You need to make sure only to meet the requirements of the application described above. How you choose to implement this application is up to you.

6. Save the modified VI.
7. Run the VI, noting the speed improvements in the Profile window and the memory use improvements in **Show VI Info...** You may 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 will prevent the LabVIEW Undo feature from allocating memory unnecessarily.

8. Once you are satisfied with the VI, save it as **Maximized Performance.vi**.

End of Exercise 7-5

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 execution time and memory use. You can then concentrate on improving those troublesome areas to enhance the overall performance of your VI.

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 with the following options: **Windows» Show VI Info** and the Profile 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 will suffer when it is accessed.

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

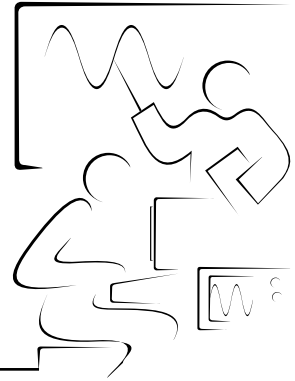
To improve the overall performance of your VIs:

- Avoid overusing local and global variables.
- Avoid displaying and manipulating large arrays and strings.
- Use functions that reuse data buffers.
- Use consistent data types (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

Lesson 8

Additional Topics



Introduction

This lesson introduces some additional features of LabVIEW and some useful programming techniques.

You Will Learn:

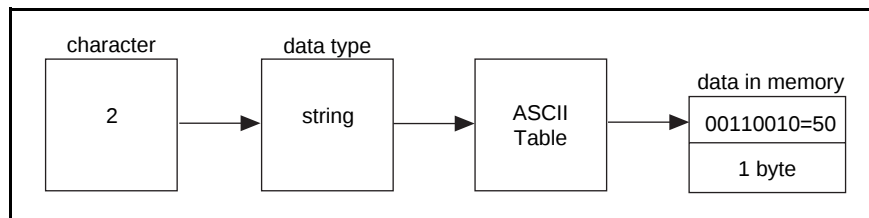
- A. How your programs manipulate LabVIEW data types.
- B. How to enhance a LabVIEW front panel with graphics and animation.
- C. How to generate run-time menus in LabVIEW.
- D. About the intensity chart and graph.
- E. About occurrences.
- F. About reentrant VIs.
- G. How to use the CVI Function Panel Converter.
- H. How to use HiQ through LabVIEW.

A. Data Manipulation Techniques

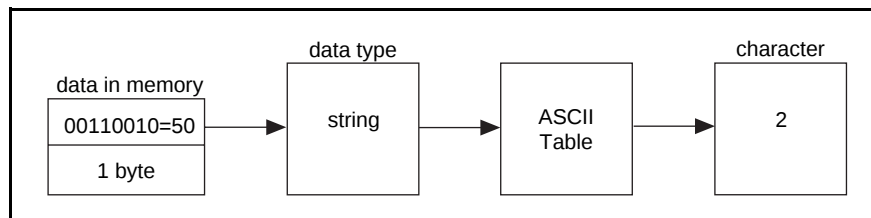
How Your Programs Interpret Data Types

Because a computer stores all information in bytes of memory, your program must know how to interpret the bytes of data you want to manipulate. You assign *data types* to your data to distinguish how your program interprets a specific byte, or grouping of bytes, of data.

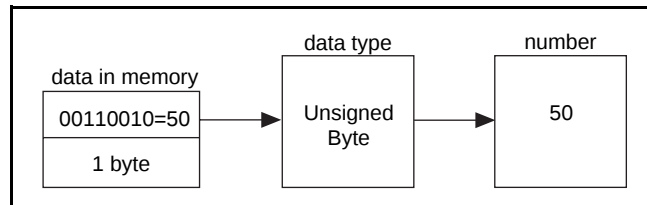
For example, recall that each ASCII text character in a string uses one byte of memory. If you use the ASCII character codes table in the Online Reference, Appendix A of this manual, or Appendix B of the *Function Reference Manual*, you will find that the character “2” in a text string has a decimal value of 50 in memory. The following illustration shows how your program uses the data type to store the character “2” in a string in memory, assuming you enter and view the data using string controls and indicators in the Normal Display mode.



The computer simply stores information in a byte of memory; it does not care what the information means. *Your program* interprets this specific byte of data as a character in a string. When your program accesses this data, it recognizes the string data type and outputs the character corresponding to the ASCII value. As shown below, your program recognizes the ASCII value of 50 and outputs the character “2”.



Now, consider having your program interpret the same byte of data as an Unsigned Byte. As shown below, your program recognizes the Unsigned Byte data type and treats the value 50 as if it were a number and nothing more.



So, the same byte of information can mean different things, depending on how your program interprets the information. The next section discusses how to convert your data between data types to change how your program interprets data.

Type Casting, Flattening, and Unflattening Data

To wire functions in LabVIEW, you must make sure that you use the correct data types. Sometimes, this involves converting from one data type to another. For example, if you try to wire a string constant containing the character “2” to the **Add** function, you will get a broken wire because the **Add** function accepts only numeric data types as inputs. To convert the string “2” into a number, you must use a string conversion function.

In LabVIEW, there are several ways to convert data from one type to another. For example, you can use the **Format Into String** and other string functions to format data into an ASCII string. These conversions involve taking input data from one type (such as a number) and creating new data of a different data type (such as a string).

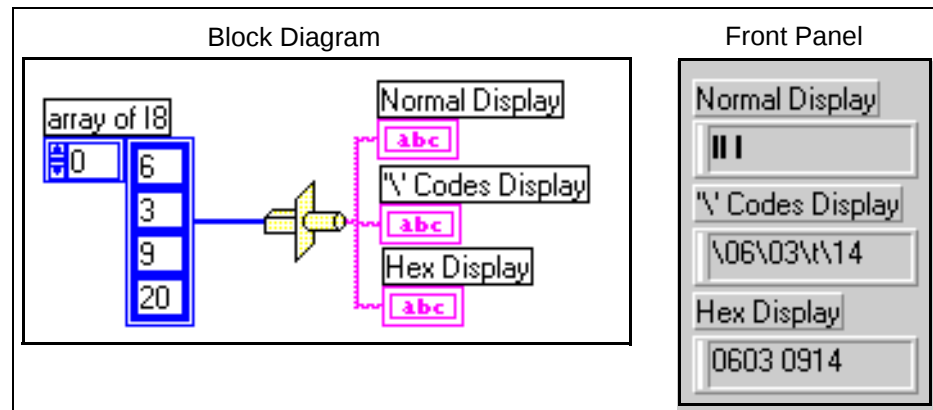
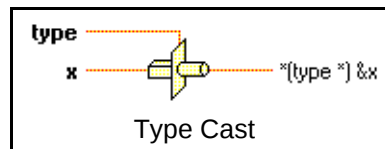
You will learn two methods for creating binary strings from your data. The **Type Cast** function changes the data type of a wire, or how your program interprets a given piece of data. However, this function does not work with all data types. The **Flatten To String** function creates a binary string given any kind of data. Its counterpart, **Unflatten From String**, converts the input string back into the original data.

Flat and Non-Flat Data

Flat data has a fixed size in memory. Examples of flat data are scalar numbers, Booleans, and clusters of scalars or Booleans. Arrays and strings are not flat. More information about LabVIEW data types is available in the Online Reference, the *LabVIEW User Manual*, and the *Code Interface Reference Manual*.

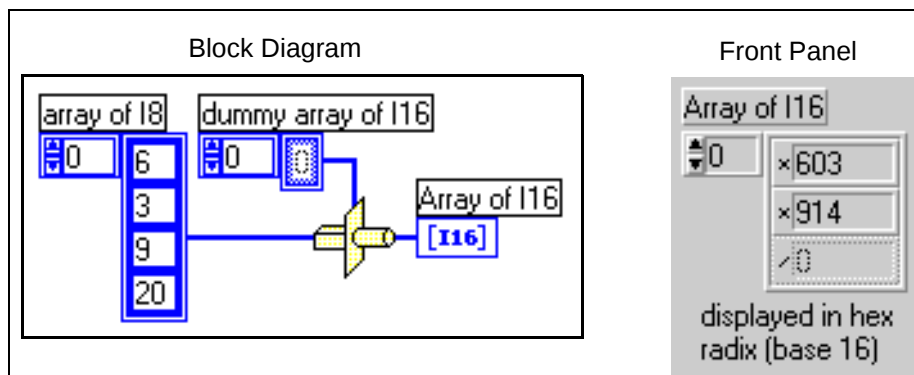
Type Casting

The **Type Cast** function (**Advanced»Data Manipulation** palette) simply changes the data type of the input data. Recall that the data type tells your LabVIEW program how to interpret the data. If you leave the **type** input unwired, the output of the **Type Cast** function is a string that contains *exactly the same bytes of data* as sent to its input. This function does *not* convert the data; it changes how LabVIEW *interprets* this group of bytes of data by changing its data type. For example, the input **x** may be an array of 8-bit integers. The output terminal views the data as a string instead of as an array of integers.

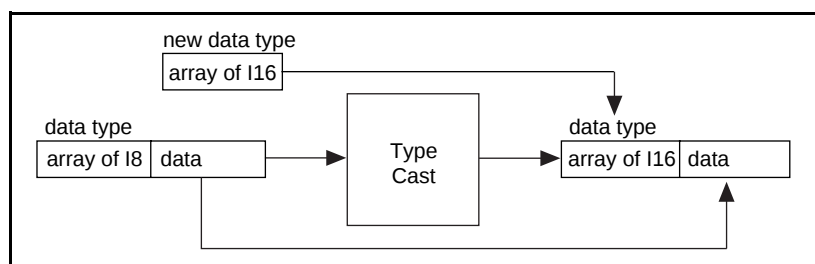


Notice the differences between the three displays shown above. The third element in the array, which has a decimal value of 9, corresponds to the <Tab> character in the ASCII character set. In the Normal Display mode, the <Tab> character is simply some space between the second and fourth non-displayable characters in the string. The '"\" Codes Display' mode shows the <Tab> character as \t, while the Hex Display mode shows the numeric value in hexadecimal notation, 09.

To change a data type to something other than a string, you must wire a dummy value of the corresponding data type to the **type** terminal of the **Type Cast** function. For example, to type cast the array of 8-bit integers into an array of 16-bit integers, connect the array of 8-bit integers to the **x** terminal, and an empty array of 16-bit integers to the **type** terminal.



The output array contains half as many elements as the input array because the number of bytes of data does not change, so two 8-bit integers are combined to form one 16-bit integer. The following illustration shows how the **Type Cast** function changes a wire's data type.

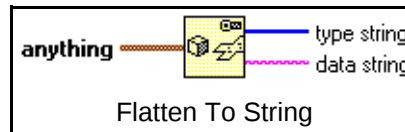


The **Type Cast** function will work with only *flat* data or one-dimensional arrays of flat data because it changes how LabVIEW interprets a *single group* of bytes in memory. You cannot use the **Type Cast** function with an array of strings as an input, nor with a two-dimensional array of numbers, because the **Type Cast** function does not include the additional information necessary to recover data originally stored in these more complex data structures.

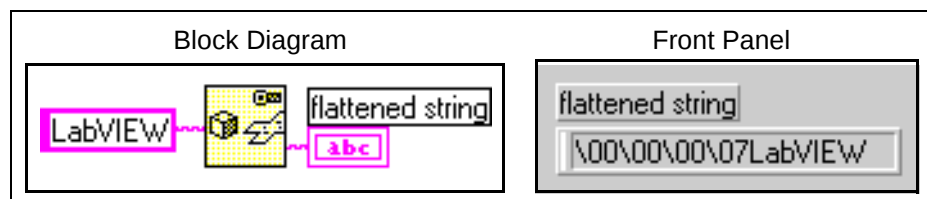
The **Type Cast** function is very useful in GPIB, VXI, TCP, and file I/O applications.

Flattening to String

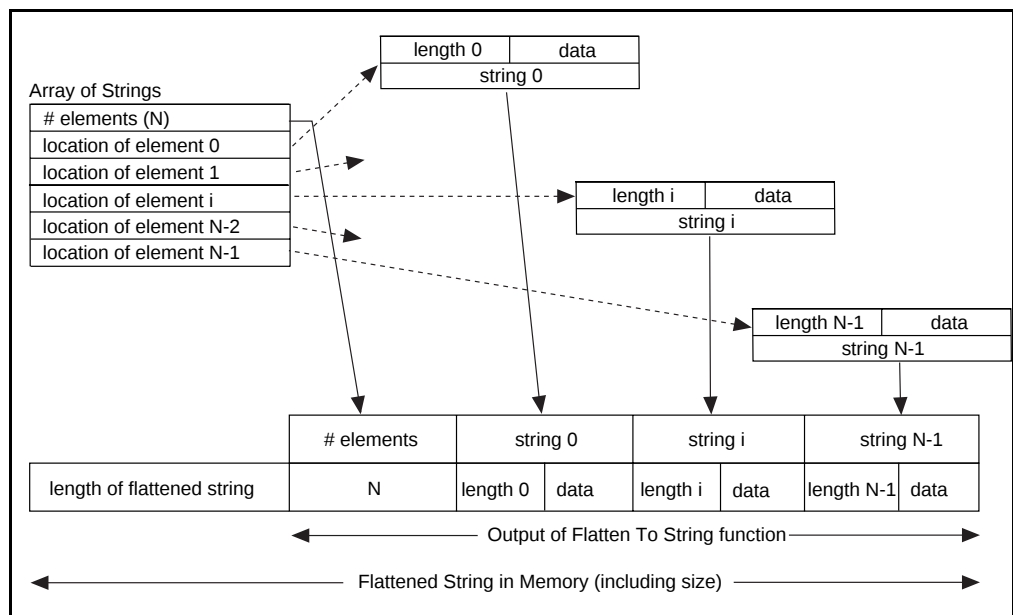
The **Flatten To String** function (**Advanced»Data Manipulation** palette) converts *any* LabVIEW data structure into a binary string. Unlike the **Type Cast** function, which changes only the type of the input data, the **Flatten To String** function gathers all of the data contained in the input, as well as any necessary header information, and creates a new output string. The header information in the output string depends on the type of data sent into the **Flatten To String** function, and it is necessary to recover the original data from the string using the **Unflatten From String** function.



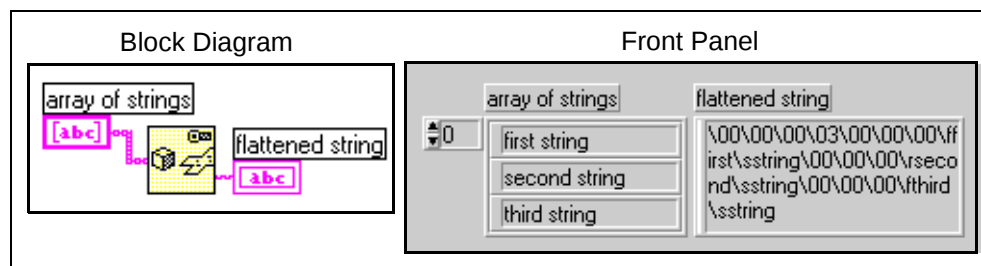
For example, when you use the **Flatten To String** function with a string input, the output data string contains not only the characters in the string, but also the four-byte header that stores the length of the string. The string indicator in the figure below uses the '\ ' Codes Display mode. Notice that the first four bytes of the flattened string contain the length of the string in bytes, which is how LabVIEW stores the string in memory.



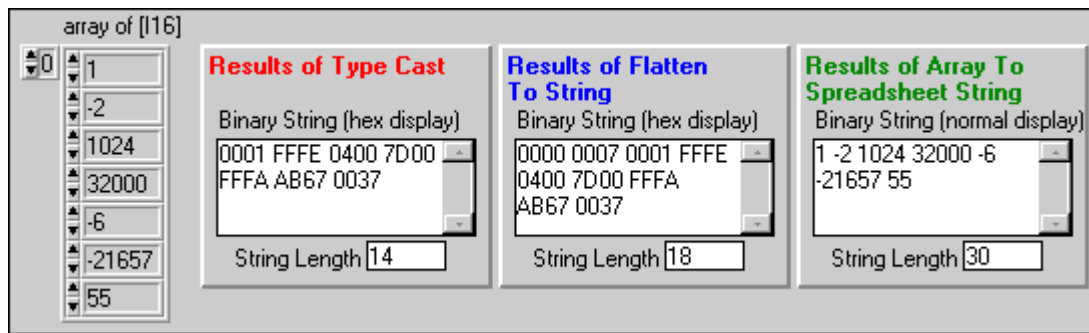
If you flatten an array of strings into a single string, the output has a special form that includes header information necessary for LabVIEW to recreate the array of strings. The **Flatten To String** function collects all of the elements in the array of strings into one large string, which occupies a contiguous block of memory. LabVIEW first stores the header information for the array—that is, the number of elements in the array of strings—followed by each string element. Each string element consists of its size in bytes followed by the data in the string. The following illustration shows how LabVIEW stores a flattened string in memory.



In the following example, the string indicator is displaying the contents of the flattened string in the '\ ' Codes Display mode. Notice that the first four bytes `\00\00\00\03` store the number of elements in the array. The next four bytes `\00\00\00\0f` provide the length of the first string, followed immediately by the characters in the string. Then, the next four bytes provide the length of the second string `\00\00\00\r`, and so on. Recall that the '\ ' Codes Display mode does not always show hexadecimal values. The `\r` code represents ASCII character 13, which is the carriage return, and `\f` is a form feed.

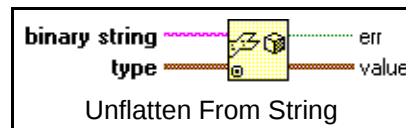


The following figure shows the results of *flattening*, *casting*, and *converting* a three-element I16 (Word) array into a string. Notice that the binary strings always use two bytes to represent each element of the array, while the ASCII string varies in length, depending on the number of characters needed to represent each number.

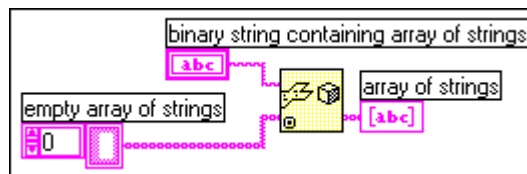


Unflattening Data from a String

The **Unflatten From String** function (**Advanced»Data Manipulation** palette) unflattens the data wired to **binary string**. This function uses the data type of the wire connected to the **type** input terminal to interpret **binary string** and to create data of the same data type at the **value** terminal. If **Unflatten From String** cannot use **type** to correctly interpret **binary string**, then the **err** output is TRUE, indicating that an error occurred when converting the data.



To use **Unflatten From String** on a string created with **Flatten To String**, the data type wired to the **type** input must be *exactly the same* as the data type of the object connected to the **anything** input of the **Flatten To String** function. The following example shows how to unflatten an array of strings from a binary string.



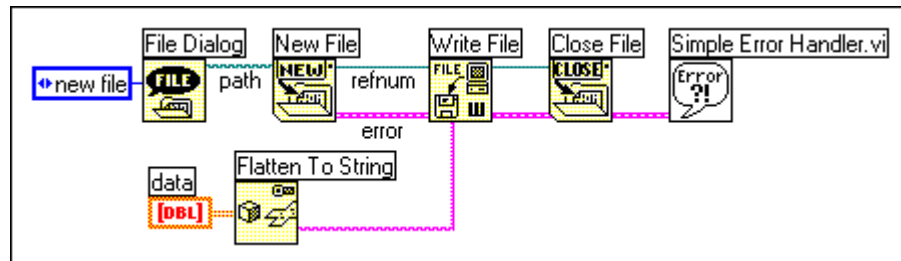
Binary Format Used for Flattened and Type Cast Data

The **Type Cast** and **Flatten To String** functions create data that is in *big endian* format. This means that for multi-byte data types, the most significant byte of a number is the first byte in the string. If your system uses *little endian* format, in which the least significant byte in a multi-byte number appears first, you may need to reverse the order of the bytes in a number when converting with the **Type Cast** or **Flatten To String** functions. The **Swap Bytes** and **Swap Words** functions are in the **Advanced»Data Manipulation** palette.

For cross-platform data transfers, LabVIEW uses the 16-byte format for extended-precision numbers when you use the **Type Cast** or **Flatten To String** functions. For example, if you type cast an array of EXT data to a string and send it from a Macintosh to a Sun workstation via a TCP connection, your programs must be able to correctly interpret the data. The Sun uses 16-byte, extended-precision numbers, while the Macintosh uses 12-byte, extended-precision numbers. To successfully recover the data, you must use a consistent representation.

Binary File I/O Using Flatten to String and Unflatten from String

You can generate a file with a header using the **Flatten to String** function. The **Flatten To String** function collects all of the data in its input terminal and places it in a binary string. It precedes the data in the output string with header information necessary to decode the string back into the original data type. The following example shows how to use the **Flatten to String** function to create a binary file with a header.

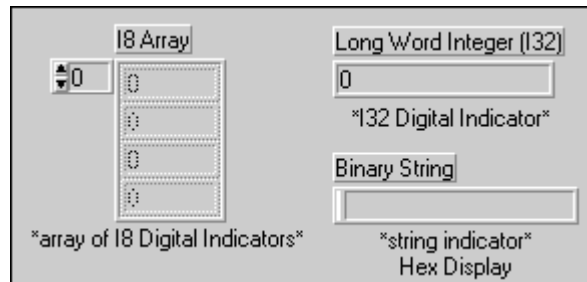


Exercise 8-1 Cast and Flatten.vi

Objective: To build a VI that uses the **Type Cast** and **Flatten To String** functions.

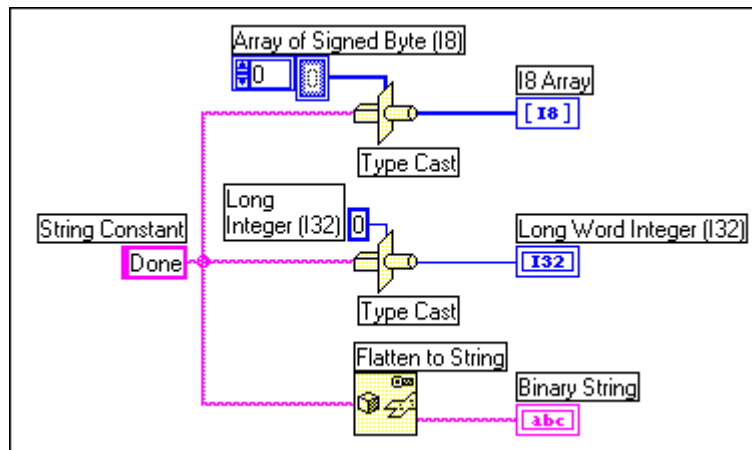
You will create a VI that uses the **Type Cast** function to interpret a string as two different types of data, and the **Flatten To String** function to observe the results of the function when used with a string input.

Front Panel



1. Open a new VI front panel and build the front panel shown above.
2. Create an array of digital indicators. Pop up on a digital indicator inside the array and select **Byte (I8)** from the **Representation** menu.
3. Create a digital indicator and set its representation to **Long (I32)** from the **Representation** menu.
4. Create a string indicator and set the display mode to **Hex Display**.
5. Open the block diagram.

Block Diagram



1. Build the block diagram shown above.



Type Cast function (**Advanced»Data Manipulation** palette) casts the ASCII string “Done” to a Byte Integer (I8) array and also to a Long integer (I32) by changing the data type.



Flatten To String function (**Advanced»Data Manipulation** palette) converts the ASCII string “Done” to a binary string.



String Constant (**String** palette) specifies the ASCII string “Done”.

2. Return to the front panel and run the VI.

The topmost **Type Cast** function changes the data type of the string to an array of one-byte integers. The lower **Type Cast** function casts the four-byte ASCII string into a Long integer, telling LabVIEW to interpret the data as one four-byte Long integer. The **Flatten To String** function places the ASCII string’s header (the string length) along with its data into a binary string.

3. Use the ASCII Character Code Equivalents table in the Appendix to confirm that the numbers in the array are the ASCII codes for “D,” “o,” “n,” and “e.” Also, use the table to confirm the last four bytes in the binary string.
4. Close the VI. Name it **Cast and Flatten.vi**.

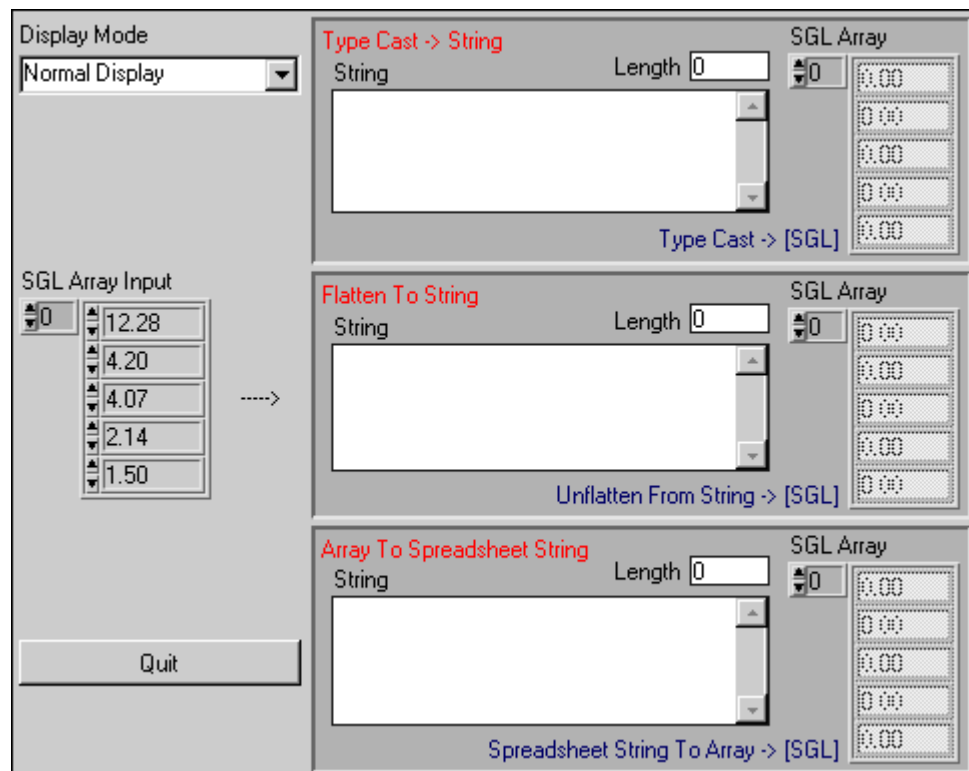
End of Exercise 8-1

Exercise 8-2 Cast, Flatten, and Convert.vi

Objective: To compare the results of the Type Cast, Flatten To String, and Array To Spreadsheet String functions.

You will run a VI that casts, flattens, and converts an array of SGL numbers to a string and then converts it back to its original representation. Notice the difference between the strings generated, as well as the length of each string.

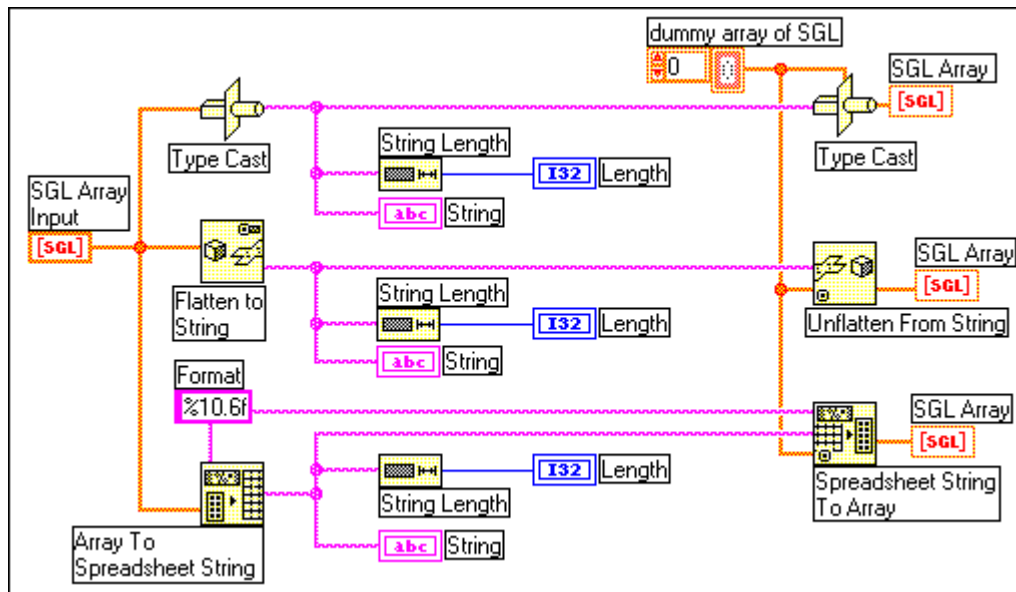
Front Panel



1. Open the **Cast, Flatten, and Convert** VI. The front panel contains an SGL array control to enter the array. The indicators are grouped together according to function pairs.

The topmost group of indicators displays the result of type casting the input array to a string, the length of the string, and the results of type casting the string back into an array of SGL data. The center group of indicators displays the result of the **Flatten To String** function when used with the input array, and the length of the string. Using the **Unflatten From String** function produces the SGL array. The bottom group of indicators shows the string created when the array is converted using the **Array To Spreadsheet String** function, the length of the string, and the array created by the **Spreadsheet String To Array** function.

Block Diagram



1. Open and examine the block diagram. The dummy array of SGL constant provides the type used for converting the string back into an array.

The casting operations are in the top third of the diagram. First, the SGL array is cast to a string using the left **Type Cast** function. The front panel displays the cast string along with its length. This string is then cast back to an array of SGL using another **Type Cast** function.

In the middle portion of the diagram, the flatten functions manipulate the data. First, the **Flatten To String** function flattens the SGL array to a binary string. The front panel displays the flattened string along with its length. The string is unflattened back to an SGL array using the **Unflatten From String** function.

The bottom third of the diagram uses spreadsheet string functions. First, the SGL array is converted to a spreadsheet string (ASCII) using the **Array To Spreadsheet String** function, which separates every element by a tab and appends an End of Line character at the end. The front panel displays the converted string along with its length. The string is then converted back to an SGL array using the **Spreadsheet String To Array** function.

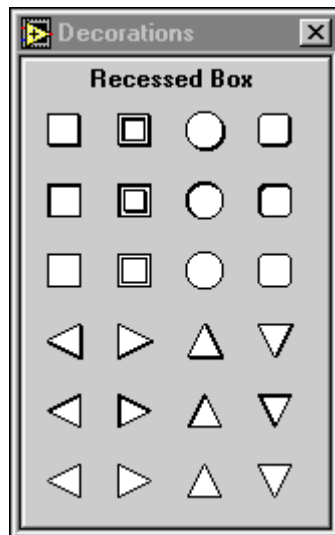
2. Run the VI. Observe the differences between the casting, flattening, and converting operations and the resulting string lengths.
3. Close the VI. Do not save any changes.

End of Exercise 8-2

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 development system, provide custom graphics and animation features to your user interface.

Decorations



One of the most straightforward methods to enhance a user interface is to apply the LabVIEW Decorations to a front panel. These objects, which you can access through the **Controls** palette, provide various flat and 3D images to serve as dividers between front panel objects. Through careful use of the decorations, you can increase the readability of your front panels.

Importing Graphics

You can import graphics from other programs 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, as described below.

- **Windows**—If you can copy an image directly from a paint program to the Windows Clipboard and then switch to LabVIEW, LabVIEW automatically imports the picture to the LabVIEW Clipboard. Or you can use the **Import Picture from File...** option from the LabVIEW **Edit** menu to import a graphics file into the LabVIEW Clipboard. LabVIEW will recognize 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 switch 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.

Once a picture is on the LabVIEW Clipboard, you can paste it as a static picture on your front panel, or you can use the **Import Picture** option of a pop-up menu, or the Import Picture options in the Control Editor.

Attribute Nodes

Every control and indicator contains the base attributes *Position* and *Bounds*. Position enables you to specify the location of an object on a front panel, while Bounds provides the size of the object. In addition, each front panel control or indicator has specific attributes that let you resize the object programmatically. By setting and reading these attributes, you can provide basic animation of a control by moving and resizing the object.

Custom Controls

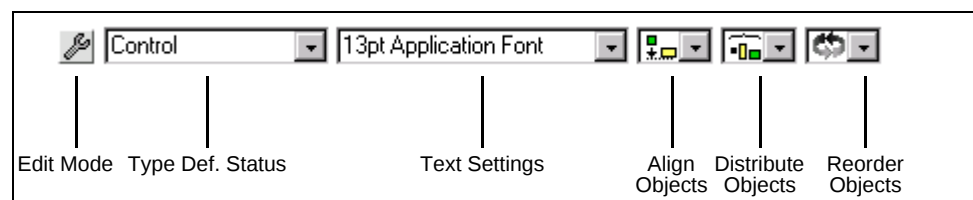
You can customize LabVIEW controls and indicators to change their appearance on the front panel. You can also save these controls for use in other VIs. Programmatically, they function the same as standard LabVIEW controls.

Control Editor

You launch the Control Editor by selecting a control on the front panel and choosing **Edit Control...** from the **Edit** menu. The Control Editor appears with the selected front panel object in its window. Just as in the LabVIEW programming environment, the Control Editor has two modes. These two modes are Edit mode and Customize mode.

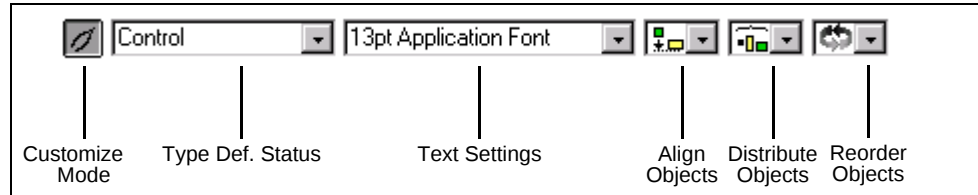
Edit Mode

Edit mode is similar to edit mode on a LabVIEW front panel. You can pop up on the control and manipulate its settings just 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 **Show Parts Window** from the **Windows** menu.



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 drop 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. You 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. You 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

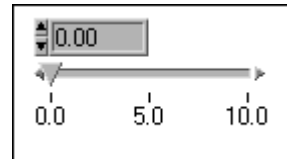
After creating a custom control, you can save it for use later. By default, controls saved on disk will have a .ctl extension. You can also place controls in the **Controls** palette using the same method as that used to add subVIs to the **Functions** palette.

You can also 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 8-3

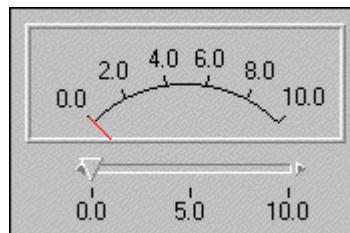
Objective: To use the Control Editor to modify a control.

1. Open a new panel.
2. Place a Horizontal Pointer Slide (**Numeric palette**) on the panel.

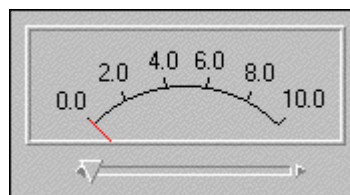


Modifying the Control

3. Launch the Control Editor by selecting the slide using the Positioning tool and choosing **Edit Control...** from the **Edit** menu. Using the Operating tool, move the slide to the middle of the panel to allow more work space.
4. Pop up on the digital display and select **Replace»Numeric»Meter**. To hide the digital display of the meter, pop up on the meter and select **Show»Digital Display**. Position the meter above the slide.



5. Hide the slide scale by popping up on the slide and selecting **Scale»Style»None**.



6. Close the Control Editor by selecting **Close** from the **File** menu. Do not save the control, but when asked to replace the existing one, reply **Yes**. The modified slider is shown on the front panel.



Note

*You can save controls that you create just as 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.

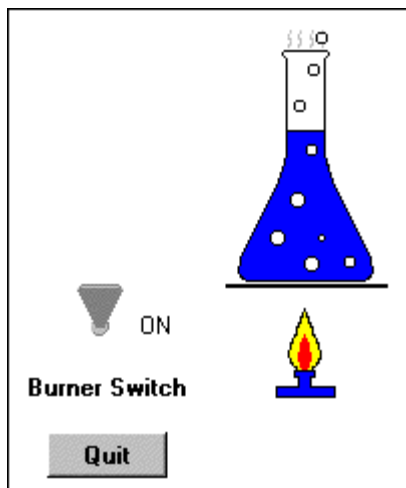
End of Exercise 8-3

Exercise 8-4 Custom Picture Exercise.vi

Objective: To create a custom Boolean indicator.

You will 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 Basics2.11b.

The VI contains a vertical toggle 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, follow the steps below.
 - a. Pop up in an open area on the front panel and choose **Square LED** from the **Controls»Boolean** pop-up palette. Label the LED *Flask*.
 - b. Using the Positioning tool, select the graphic that shows the contents of the flask boiling, and choose **Cut** from the **Edit** menu. Click on the Flask LED indicator and choose **Edit Control...** from the **Edit** menu. The Control Editor should now appear with the Flask LED displayed. Pop up on the LED and select **Import Picture»True**. This custom picture now represents the TRUE state.



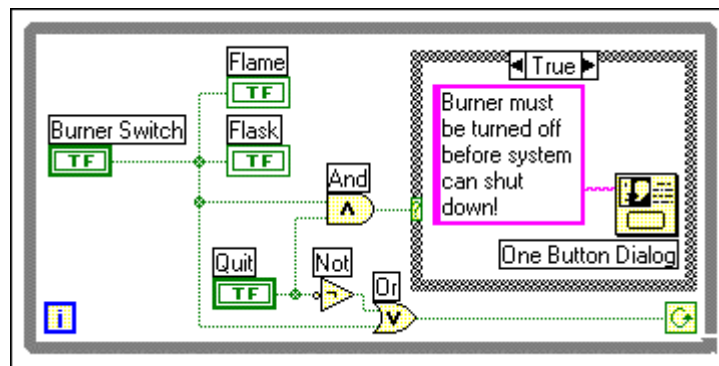
Note

The default state of the LED is FALSE. If you do not see the picture, the LED is probably in the FALSE state.

- c. Switch to the front panel by clicking on it. Using the Positioning tool, select the graphic of the flask that shows the contents of the

- flask not boiling, and choose **Cut** from the **Edit** menu. Switch to the Control Editor window by clicking on it.
- d. Pop up on 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. Pop up in an open area and choose **Square LED** from the **Boolean** pop-up menu. Label the LED *Flame*.
4. Using the steps above, 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 popping up on them and selecting **Show»Label**. Select both Boolean indicators and align them on horizontal centers using the **Align Objects** tool.

Block Diagram



1. Complete the block diagram as shown above.
2. Return to the front panel and run the VI. Turn the Burner Switch on and off and notice the custom Boolean change.
3. Stop the VI by clicking on the Quit button. If you press 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.
4. Save and close the VI.

End of Exercise 8-4

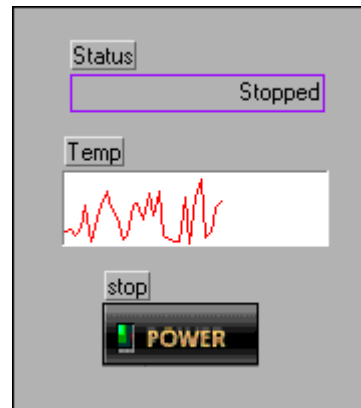
Exercise 8-5 Graphic Panel.vi

Objective: To use some of the LabVIEW built-in features to make an enhanced front panel.

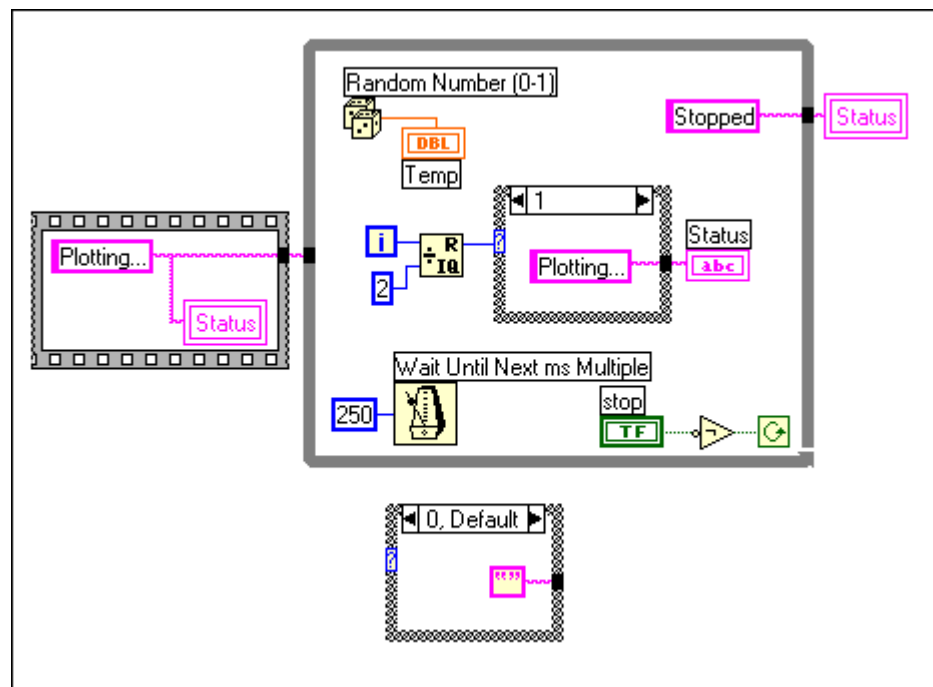
In this exercise, you will enhance a basic VI front panel using imported bitmaps and images.

1. Open the **Graphic Panel** VI located in **Basics2.11b**. A basic front panel and block diagram are already complete.

Front Panel



Block Diagram



2. Run the VI. Every 0.25 second, a random number is generated and plotted on the waveform chart. In addition, the Status string flashes the word “Plotting...” while the VI is running. The Power button is a Boolean control that has been modified using the Control Editor to have a custom bitmap image. Press the **Power** button to stop the VI.
3. Examine the block diagram. Inside the While Loop, the random number is generated, and the Quotient/Remainder function determines what string should be sent to the Status indicator. You may want to run the VI in execution highlighting mode to determine how it operates. Stop the VI if it is running and switch back to the front panel.
4. *Windows*—Expand the front panel. From the LabVIEW **Edit** menu, select **Import Picture from File....** In the dialog box that appears, find the bitmap file Panel1.bmp (which should be in the same directory as your Basics2.llb library), select it, and then click on **Open**. The bitmap image is then copied into the LabVIEW clipboard.

Click in an empty area of the VI’s front panel and then select **Paste** from the LabVIEW **Edit** menu. The bitmap of an instrument’s face should appear on the front panel.

5. Hide the owned labels for the Status, Temp, and stop controls and move them onto the instrument face bitmap.
 - The **Status** string should go to the gray towards the top of the bitmap.
 - The **Temp** chart should move to the white area of the bitmap.
 - The **stop** button should be located in the bottom portion of the bitmap.

You may want to use the arrow keys on the keyboard to accurately position the objects. Also, when you move these objects onto the bitmap, they may move behind the image. If this happens, use the **Reorder** ring control and select **Move to Front**. This will place the objects on top of the bitmap, rather than behind it.



Reorder
ring control

6. Select a **Raised Frame** from the **Controls»Decorations** palette and stretch it over the boundary of the instrument face bitmap. This will finish the front panel, which should now resemble the following:



7. Resize the front panel window to reduce the amount of empty panel space and save the VI. Run the VI and observe its operation.
8. Close the VI when you are finished.

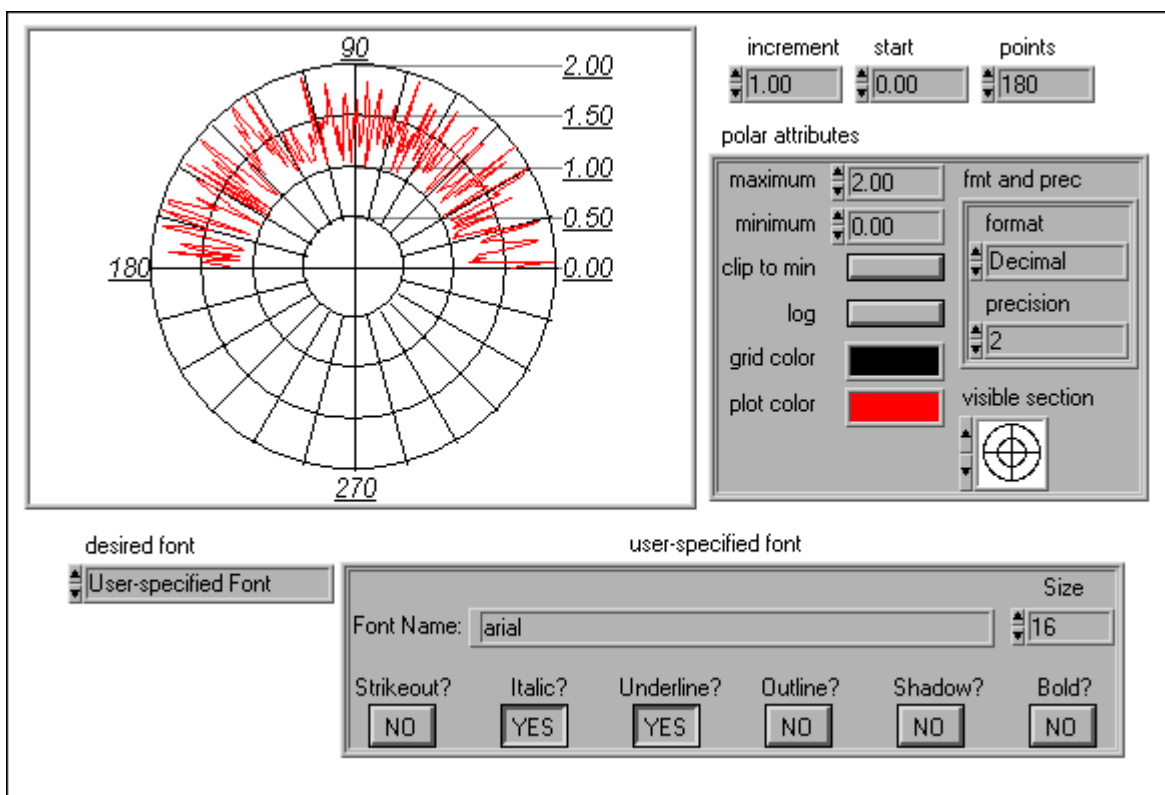
End of Exercise 8-5

The Picture Control

The Picture Control is a set of tools for creating arbitrary front panel displays. The Picture control displays picture images described by graphics instructions. The Picture Control is in the **Graph** palette. The Picture functions in the **Graphics & Sound** palette implement a common set of drawing commands for building the graphic images.

One of the best aspects of the Picture Control is its examples, which include high-level graph virtual instruments (VIs) for use “as is” in your application. Examples include VIs for creating waveforms, XY and multi-xy displays with scales and grids, and a variety of plot styles (bar, point, line, and so on). There are also specialized examples for Polar plots, Smith plots, and Waterfall plots. In addition, there is a Robot Arm example that demonstrates animation of real-world processes.

If you build off the examples, you will find many valuable subVIs for drawing waveforms in a variety of styles, creating grids, and specifying scale range and format. There are also custom controls for common graphics objects to aid in the creation and passing of picture data.



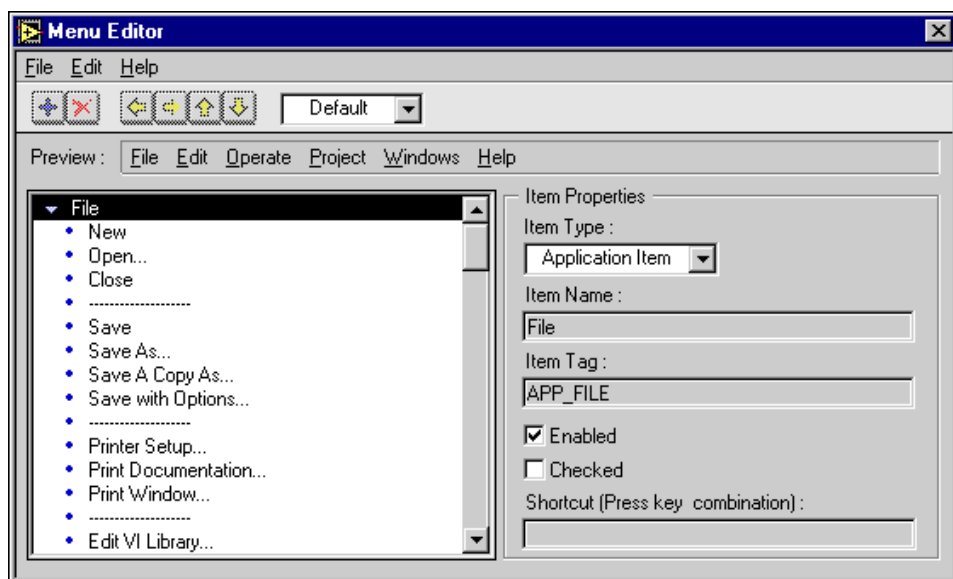
C. LabVIEW Run-Time Menus

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 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 easily 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 invoking the **Edit Menu...** option from the **Edit** pull-down menu in LabVIEW. Invoking the Menu Editor brings up a dialog box, as shown below:



On the left side of this window 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 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 diagram. For ease in editing, whenever 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 on the arrow button next to the Item Name and follow the pop-up hierarchy as shown in the illustration. 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 diagrams. You cannot alter the name, tag, or other attributes of an Application Item. LabVIEW reserves tags starting with “APP_” for Application Items. The tags for Application Items are listed in the LabVIEW Online Reference.

A **Separator** inserts a separation line in the menu. You cannot set any of the attributes 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 attributes. You can set a shortcut (accelerator) for a User Item by selecting an appropriate key combination.

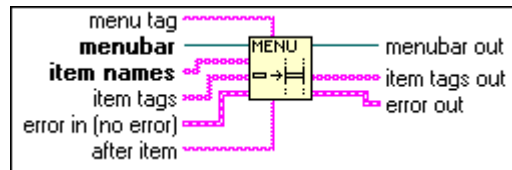
The Menu Editor allows you to insert, delete, or reorder menu items in the menu hierarchy. Clicking on the “+” button adds a new menu item. You can also change the type of a menu item by choosing from the **Item Type** ring. In addition, you can reorder the menu items and create submenus using the arrow 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 just edited.

Dynamic Menus

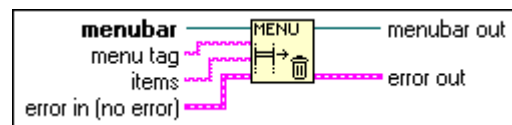
While a VI executes, you can change items in the VI menu bar by using the menu management functions described below. All of these functions are located under **Function»Application Control»Menu**. All of these functions operate on a refnum for the menu retrieved using the **Current VI's Menu** function.

Insert Menu Items



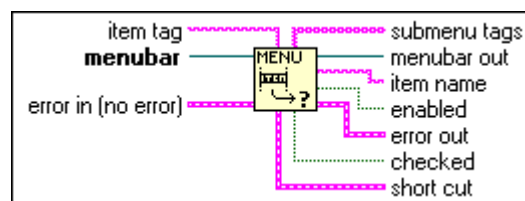
Inserts a menu item (or an array of menu items) identified by **item names** or **item tags** into a menu (menubar or submenu). The position of the item is specified either through the **after item** parameter or through a combination of the **menu tag** and **after item** pair.

Delete Menu Items



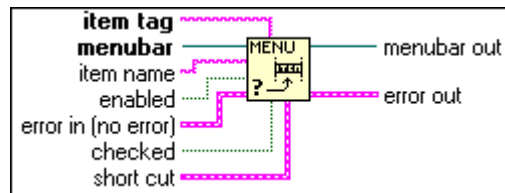
Deletes a menu item (or an array of menu items) identified by **items** from a menu identified by **menu tag** (from the menu bar, if **menu tag** is not specified). **Items** can be a tag, array of tags, position number, or an array of position numbers.

Get Menu Item Info



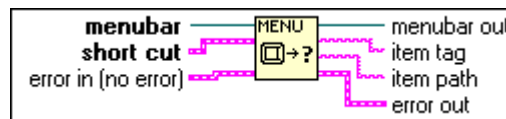
Returns attributes of the menu item specified through **item tag** (or the **menubar** if the **item tag** is unspecified). Item attributes include **item name**, **enabled**, **checked**, and **shortcut**. If the item has a submenu attached, its item tags are returned in **submenu tags**.

Set Menu Item Info



Sets attributes of the menu item specified through item tag. Item attributes include item name, enabled, checked, and shortcut. Unwired attributes 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 Setup or dynamically during run time. To set up the diagram to handle menu selections, acquire control over the menu selection process with the function **Get Menu Selection**. 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**.

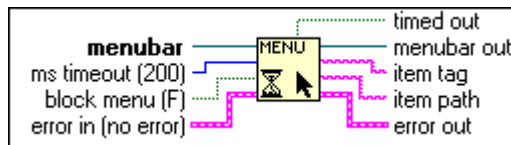
Once you select an item, you cannot select another item until the **Get Menu Selection** function reads the first item. Under such conditions, **Get Menu Selection** is invoked under block menu mode, wherein menu tracking is blocked out after a selection is read. The menu is enabled after you process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

Current VI's Menubar



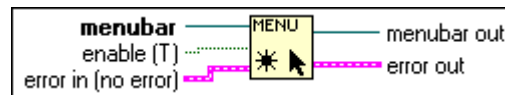
Returns the **menubar** refnum of the current VI. This function must be executed before the other menu handling functions are invoked.

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



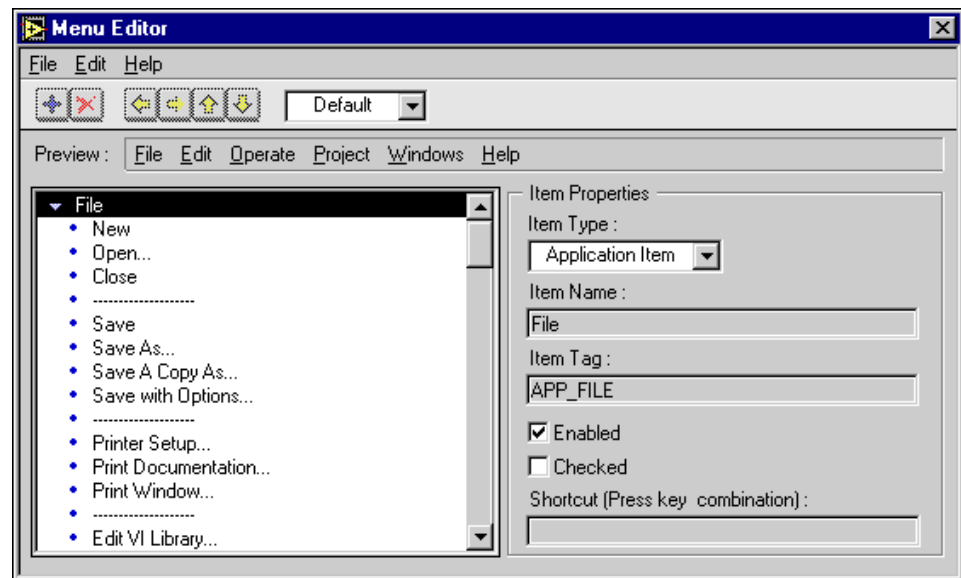
This function enables or disables the tracking of menus. Once a menu is blocked using the **Get Menu Selection** function, **Enable Menu Tracking** must be executed to re-enable the pull-down menu.

Exercise 8-6 Pull-down Menu.vi

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 **Basics2.11b**. 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 may access this menu.
2. From the LabVIEW **Edit** menu, select **Edit Menu....** The Menu Editor should appear:



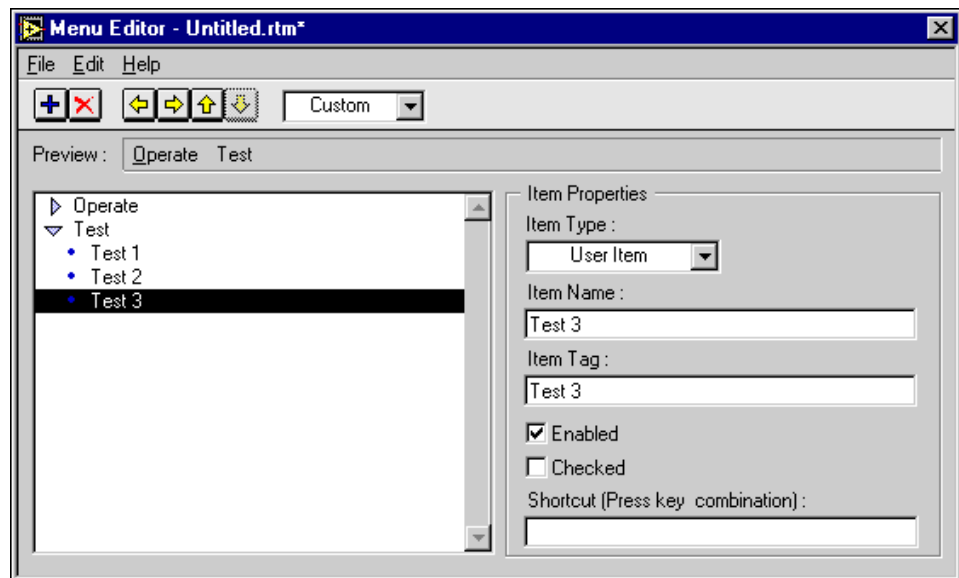
The current run-time menu for the application is the LabVIEW default menu. In the next several steps, you will replace that menu with a custom list of selections.

3. Change the topmost ring control in the Menu Editor from **Default** to **Custom**. The menu listed on the left portion of the window 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 on the triangle next to the

Operate option. You should now see only the **Operate** item in the menu list.

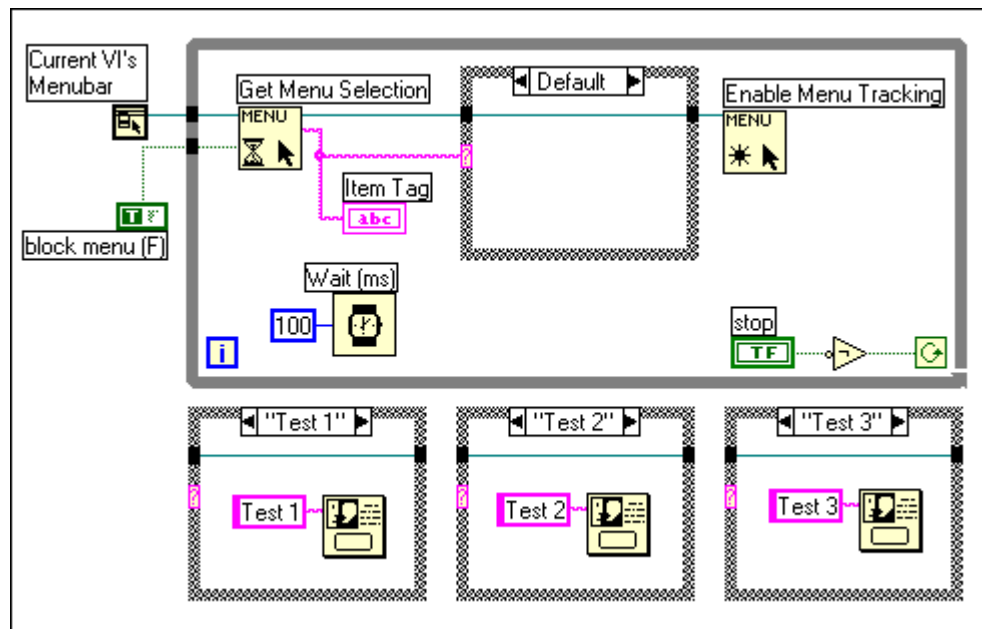
5. Click on the “+” button in the Menu Editor toolbar. A new unnamed item (“???”) will appear in the menu list. With this item highlighted, enter **Test** into the **Item Name** property. This menu item will now have an Item Name and Tag of **Test**.
6. Click on the “+” button again to add another entry under the **Test** item. Press 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 window 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. From the Menu Editor **File** menu, select **Save**. Save the run-time menu as **Menu Exercise.rtm** in the **C:\Exercises\LV_Bas2** directory. Then close the Menu Editor window. When LabVIEW asks if you want to change the run-time menu to **Menu Exercise.rtm**, select **Yes**. You have just configured a custom pull-down menu which will be invoked while the VI executes.

Block Diagram



9. Switch to the block diagram of the VI and complete it as shown above.



Current VI's Menubar (Application Control»Menu subpalette).

This function returns the refnum for the selected VI's pull-down menu, so that it may be manipulated.



Get Menu Selection (Application Control»Menu subpalette). 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.



Enable Menu Tracking (Application Control»Menu subpalette).

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 panel window. 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 pops up 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 press the OK button on the dialog box, the Item Tag indicator is cleared and the menu is re-enabled by the **Enable Menu Tracking** function. Also, notice that the items from the **Operate**

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 may want to turn on execution highlighting and single stepping and examine the block diagram. Press Stop on the VI's front panel to halt program execution.

11. 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>	< ⌘ -1>
Test 2	<Ctrl-2>	< ⌘ -2>
Test 3	<Ctrl-3>	< ⌘ -3>

In addition to setting the above 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_execute. In this exercise, assign ALT-T to the Test option in the menu.

12. Save the VI with the updated keyboard shortcuts and run the VI. Now you should be able to use the keyboard shortcuts, rather than the mouse, to select the different test options.
13. Close the VI.

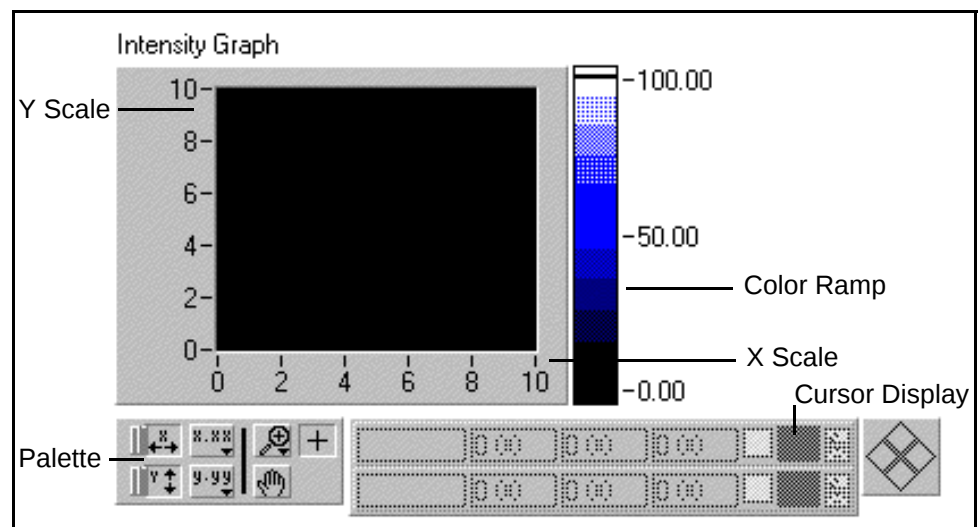
End of Exercise 8-6

D. 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 easily 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

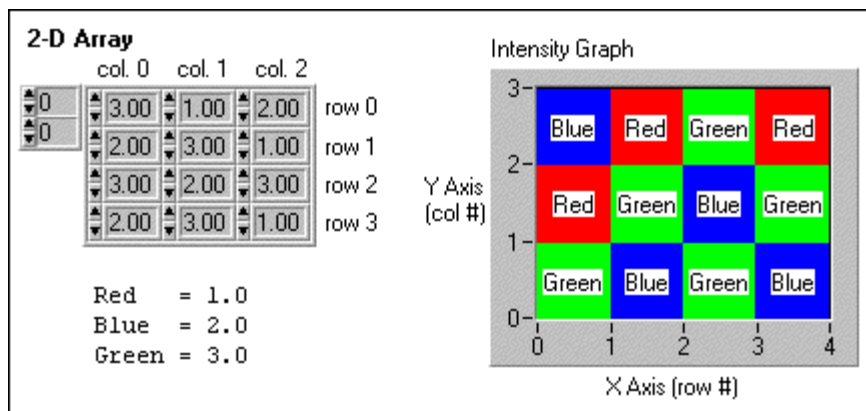
The displays of intensity charts and graphs use many of the same options as their waveform counterparts. The figure below shows many of the intensity graph options. Intensity plots also provide options unavailable in other charts and graphs. Because they display a third value (color), a color ramp is 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, pop up on 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, pop up on 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 text tool and type in the location of the marker. After you have placed the marker at the appropriate location on the ramp, pop up on 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 below shows a 4 x 3 array plotted on an intensity graph. The colors mapped are red (1.0), blue (2.0), and green (3.0).

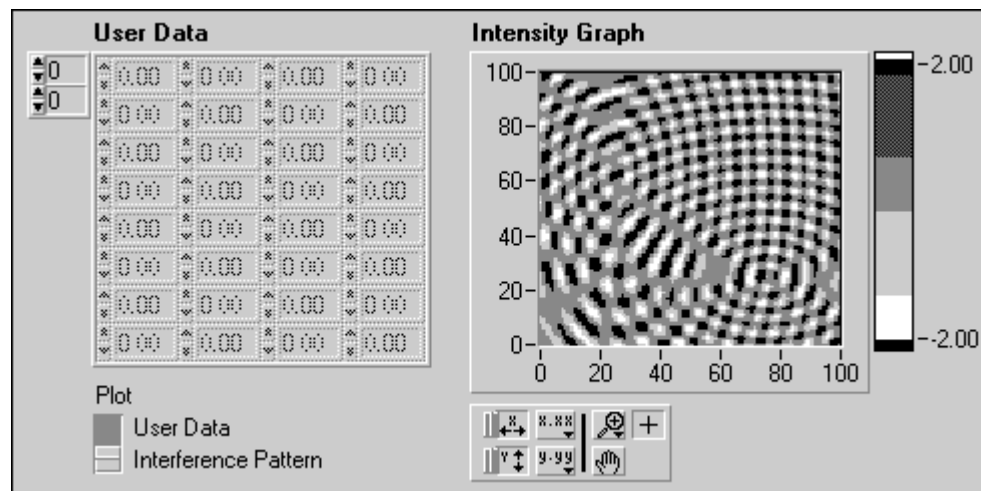


Exercise 8-7 Intensity Graph Example.vi

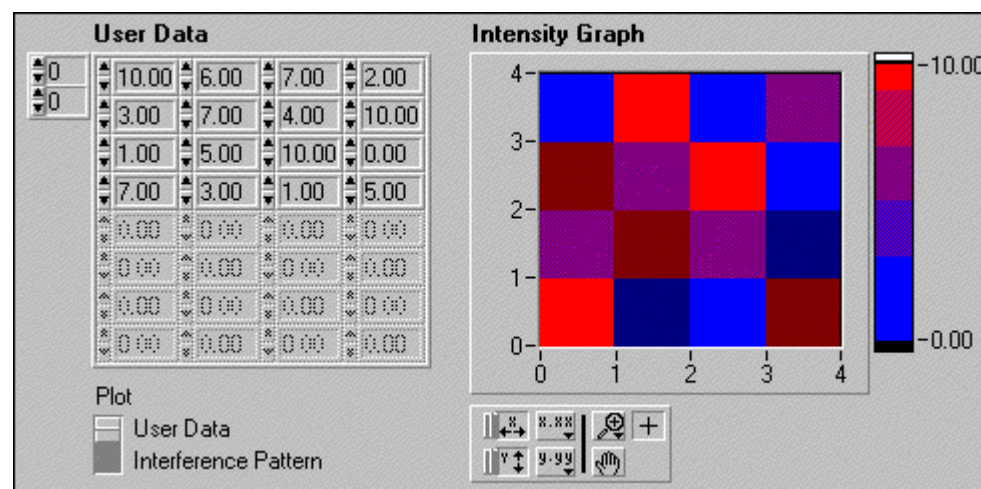
Objective: To use an intensity graph.

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 Basics2.11b. By default, the VI plots an interference waveform. An attribute 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.

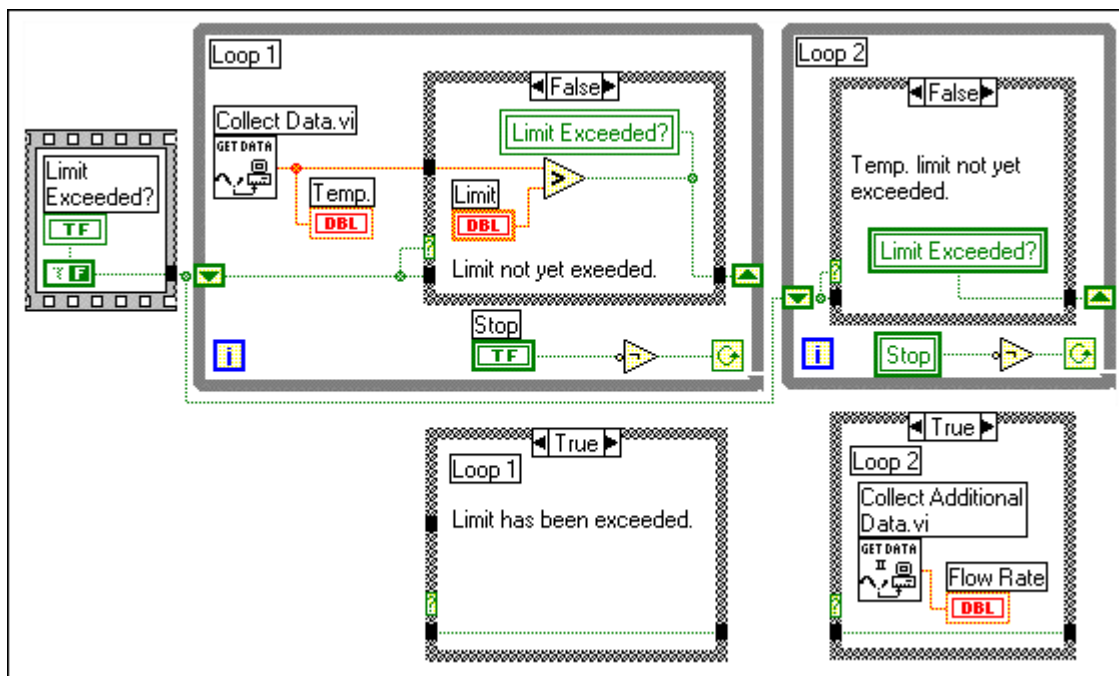


3. Close the VI. Do not save changes.

End of Exercise 8-7

E. Occurrences

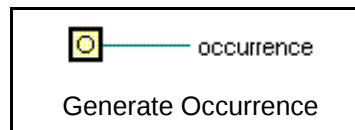
Consider an application where you want to put part of a diagram to “sleep” until a certain condition is true. Consider that you want to monitor the temperature and flow rate of a fluid through a system. However, you do not need to collect the flow rate information until after the temperature has exceeded a certain limit. After it has, your application should continue to measure both temperature and flow rate. As shown below, you could use two independent loops and local variables to control the data collection. The two loops use shift registers to store whether or not the temperature limit has been exceeded. After it has, the value in each shift register remains unchanged.



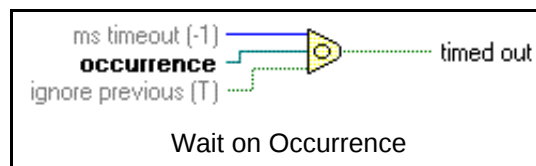
Consider an alternative approach. You can use occurrence functions to control separate, synchronous activities. In particular, you use these functions when you want one part of a block diagram to wait until another part of the same block diagram finishes a task without forcing LabVIEW to poll.

LabVIEW uses three functions to control occurrences, which you can find in the **Advanced»Synchronization»Occurrences** palette of the **Functions** palette.

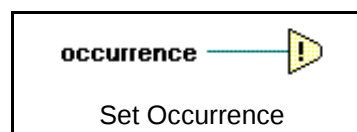
Generate Occurrence creates an occurrence that you can pass to the **Wait on Occurrence** and **Set Occurrence** functions. The Occurrence output is a refnum that links **Wait on Occurrence** and **Set Occurrence**.



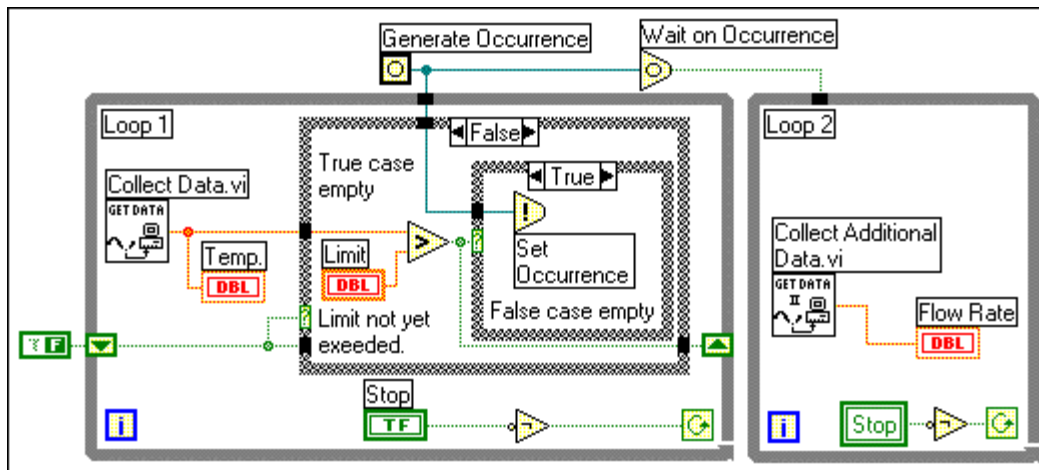
Wait on Occurrence waits for the **Set Occurrence** function to trigger the given occurrence. You can use the **ms timeout** input to force this function to finish after the given amount of time passes, regardless of whether the occurrence was set elsewhere. If this happens, the **timed out** output value is TRUE. The default value of the timeout is -1, meaning that the **Wait on Occurrence** function will wait indefinitely.



Set Occurrence triggers the specified occurrence. All block diagrams waiting for this occurrence stop waiting, or “wake up.”



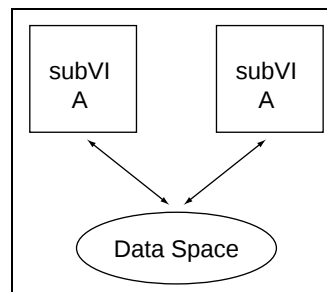
The diagram of a VI that implements the same application as the previous example appears below. The **Generate Occurrence** function provides a unique occurrence reference number that is sent to both Loop 1, which measures temperature, and the **Wait on Occurrence** function. The output of the **Wait on Occurrence** function is wired to the border of Loop 2, which reads the Flow Rate. Because there is a wire connecting the output of the **Wait on Occurrence** function to the border of Loop 2, Loop 2 cannot start until the **Wait on Occurrence** finishes. Recall that **Wait on Occurrence** finishes when the **Set Occurrence** function executes. The occurrence is set in Loop 1 the first time the measured temperature exceeds the limit value.



F. Reentrant VIs

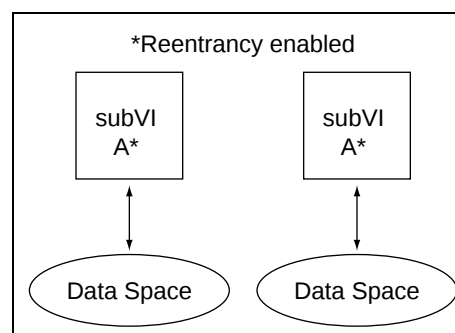
In some instances, you may build a VI that uses *uninitialized* shift registers to store data. If you plan to use that VI as a subVI and make multiple calls to it, you must be careful that the calls do not share the same data.

For example, consider a VI “A” that uses uninitialized shift registers to store data. As shown to the right, two calls are made to “A.” Both calls share the same data space. This could cause a problem if the VIs are meant to execute independently of each other.

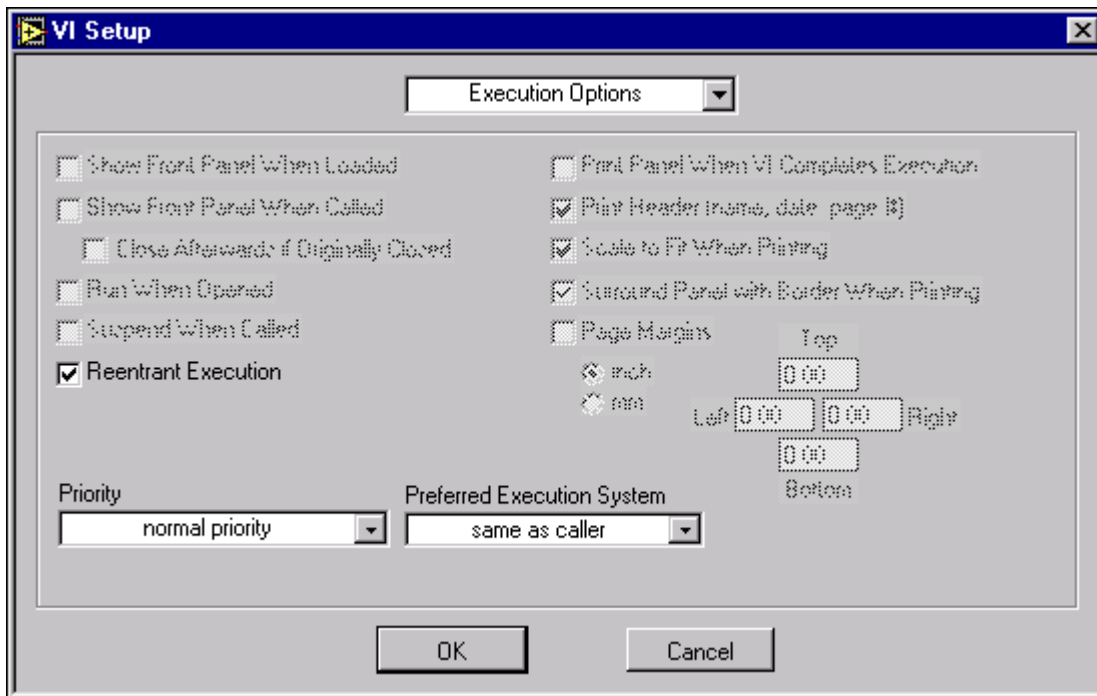


One solution to prevent the calls from sharing the same data space is to save several copies of “A,” each with a different name (for example, “A1,” “A2,” and so on.). However, this method is cumbersome if many calls are made to the VI. You also unnecessarily duplicate code, thus using more memory.

A better solution is to make the VI *reentrant*. If a VI is made reentrant, each instance of the VI in memory uses a different data space. This is illustrated at the right. Because reentrancy is enabled, each call to “A” will use a different data space, allowing each call to execute independently of the other without the danger of sharing the data. Each instance of a reentrant VI executes in parallel, but the code for the VI is loaded into memory only once.



You enable reentrant execution by popping up on the icon pane in the Panel window and choosing **VI Setup...** from the pop-up menu. Notice that if reentrancy is enabled, several other options are disabled.



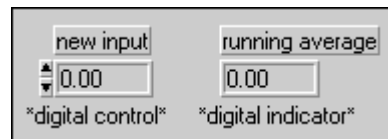
You will now do an exercise that uses reentrancy to keep two sets of averaged data from mixing previously stored values. The effects of reentrancy on memory are discussed in the Memory Management module of the LabVIEW Advanced I course.

Exercise 8-8 Running Average.vi

Objective: To build a reusable four-point running average VI.

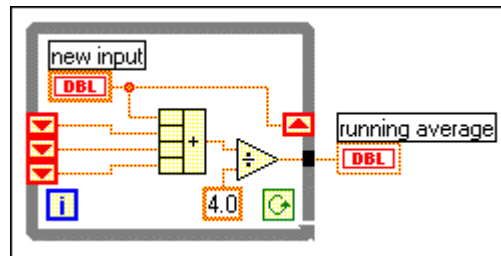
You will build a VI that calculates a running average of four data points. The VI will use an uninitialized shift register (with three additional elements) to store the four data points. By making this VI reentrant, you can use it in any VI diagram.

Front Panel



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

Block Diagram

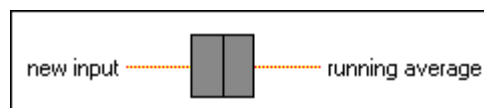


1. Build the diagram shown above.
2. Create the icon and connector for the VI.

Icon



Connector



3. Make the VI reentrant by popping up on the icon pane in the Panel window and choosing **VI Setup...** from the pop-up menu. Configure the VI to be reentrant, as shown in the VI Setup dialog box on the previous page.
4. Save and close the VI. Name it **Running Average.vi**. You will use the VI as a reentrant subVI in the next exercise.

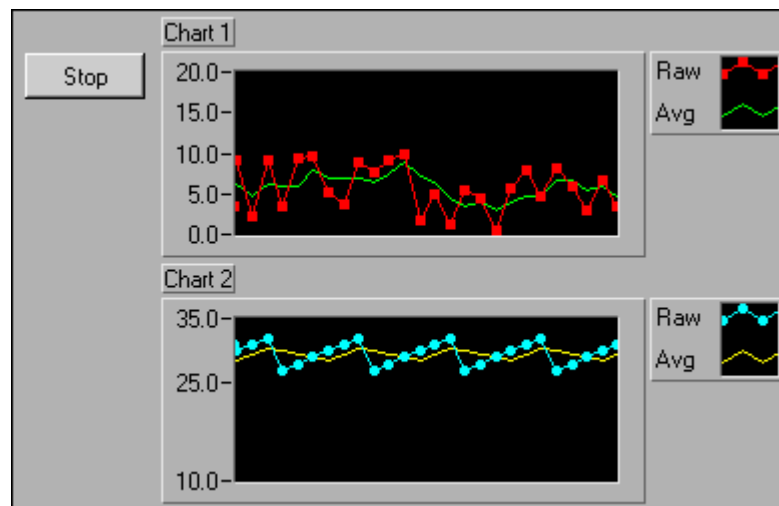
End of Exercise 8-8

Exercise 8-9 Running Avg Application.vi

Objective: To use a reentrant subVI.

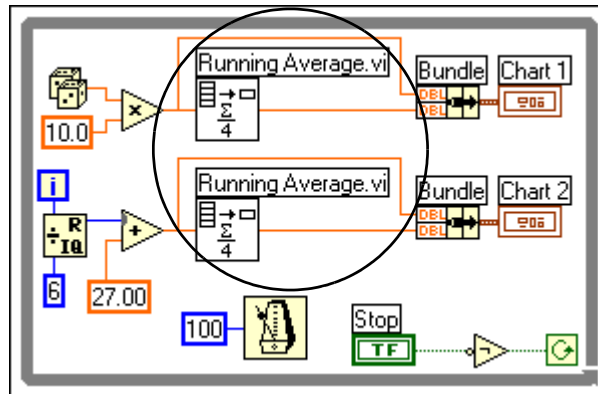
You will build a VI that calculates and plots the running average of generated data points. The VI calculates the running average using the **Running Average** VI that you built in the previous exercise. Because the VI is reentrant, each call to the **Running Average** VI uses its own data space.

Front Panel



1. Open the **Running Avg Application** VI in Basics2.11b. The front panel of the VI is already created. You will finish building the block diagram.
2. Open the block diagram.

Block Diagram



1. Build the portion of the block diagram shown in the ellipsis above.



Running Average.vi (Select a VI... menu) calculates the average of the last four random numbers.

2. Save the VI and return to the front panel.
3. Run the VI for a short time. After stopping it, you will modify the reentrancy of the **Running Average** VI to observe the effects of reentrancy.
4. Open the block diagram. Double-click on the Running Average icon. Its front panel should open.
5. On the front panel of **Running Average.vi**, disable the reentrancy option of the VI in the **VI Setup** dialog.
 - a. Pop up on the icon pane of the Panel window and choose **VI Setup...** from the pop-up menu.
 - b. Click in the box next to “Reentrant Execution” to disable it.
6. Close all windows and return to the **Running Avg Application** front panel. Do not save any changes. (The changes you make to the **Running Average** subVI remain in memory.)
7. Run the VI. Notice that the averages are no longer correct because the data space is being shared. Some of the data displayed on Chart 1 is used in calculating the running average shown on Chart 2, and vice versa.
8. Close the VI. Do not save any changes.

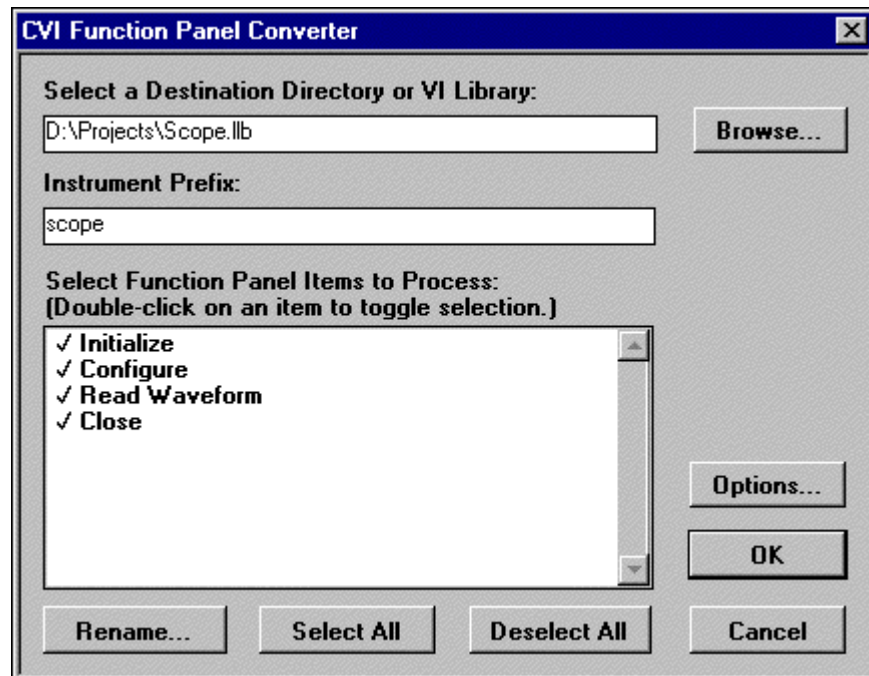
End of Exercise 8-9

G. The CVI Function Panel Converter

As you and your co-workers develop applications, some may find that working in a more traditional C programming environment better suits their application development needs. National Instruments offers such a package, called LabWindows/CVI (C for Virtual Instrumentation) for Windows and Sun. LabWindows/CVI provides an interactive, easy-to-use development environment in which you create applications in ANSI C.

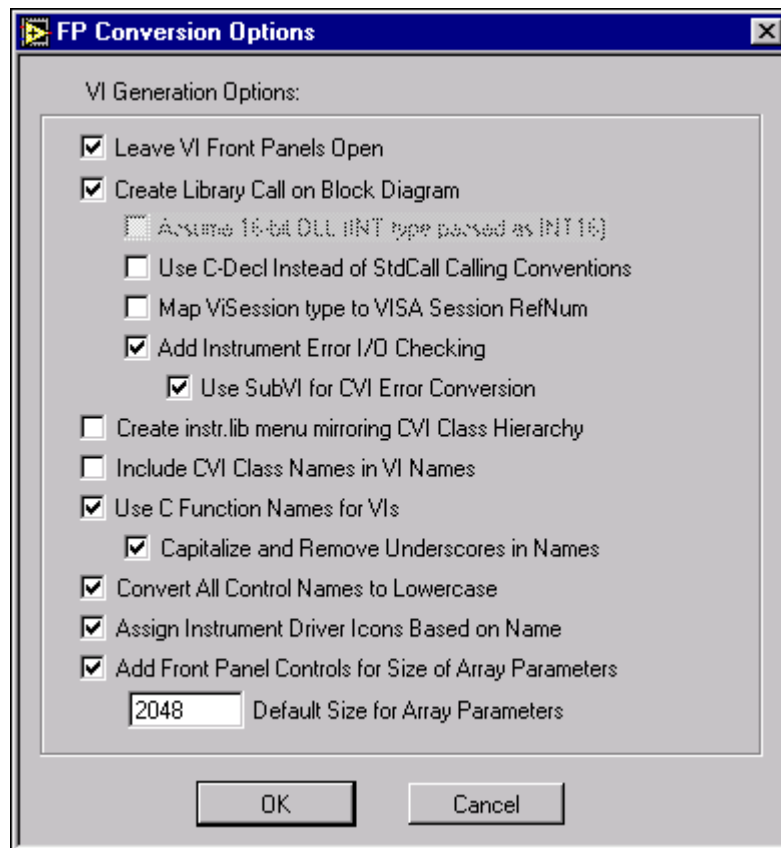
With the CVI Function Panel Converter, you can convert CVI function panels into LabVIEW VIs. To do this, you need to compile the CVI application as a Windows Dynamic Link Library (DLL) in Windows or a Shared Library on the Sun. You can create LabVIEW VIs from the CVI function panel (.fp) file that executes the code stored in the library. Thus, instrument drivers that may not be available in LabVIEW, but which a co-worker or other vendor may have developed in CVI can be made available in LabVIEW. This portability can save a great deal of time and money during application development.

To convert a CVI function panel into a LabVIEW VI, select **Convert CVI FP File...** from the **File** menu. A dialog box prompts you to find a .fp file to convert. After you have selected a file, you will see a dialog box similar to that shown below.



A single .fp file can contain many functions. You select which of these functions to convert into LabVIEW VIs in this dialog box, as well as the name of the LabVIEW VI library (.LLB file) in which the newly created VIs

will be stored. There are also a number of options you can select to customize the conversion process when you click the **Options...** button. The **FP Conversion Options** dialog box is shown below.



Enabling the **Leave VI Front Panels Open** option will show the panels of the VIs created by the CVI Function Panel Converter as they are made.

If you do not already have the Shared Library or DLL for the CVI function panels, disable the Create Library Call feature in the **Conversion Options** dialog box. The converter will create VIs with all of the necessary front panel controls and indicators as well as an icon and a connector pane. You will need to finish the block diagrams of the VIs. All CVI function panel descriptions are also placed in the LabVIEW VI front panel.

If you enable the **Create Library Call** feature, each LabVIEW VI's block diagram will contain a Call Library Function icon that is already configured to call the correct function in the library with the correct arguments and function return value. All of the terminals to the Call Library Function will also be wired.

Exercise 8-10 Scope.llb

Objective: To use the CVI Function Panel Converter.

You will create a library of VIs from a CVI function panel file.

1. Make sure you have a LabVIEW VI window open. Then, select **Convert CVI FP File...** from the **File** menu.
2. A file dialog box appears, prompting you to find a CVI .fp file. Select the Scope .fp file in the LABVIEW directory and press the **Open** button.

The **CVI Function Panel Converter** dialog box appears, similar to that shown previously in this section. You will convert all of the function panels in the file into VIs.

3. Press the **Options...** button to enter the **FP Conversion Options** dialog box. Enable the Leave VI Front Panels Open option. Click the **OK** button to exit the dialog box.
4. Press the **OK** button in the **CVI Function Panel Converter** dialog box to begin the conversion. You will notice that as each CVI function panel is converted into a VI, the front panel for the VI is opened.
5. When prompted to **Select a Library**, select Scope .dll (Scope .so on Sun) in the LABVIEW directory. This file is in the same directory as Scope .fp.



Note

If you cannot find this file, press the Cancel button in the file dialog box. The CVI Panel Converter will create the VIs as if the DLL or Shared Library were available, but you will not be able to run the VIs.

6. After completing the conversion, the converter generates a text file called scope .out. This file contains information about the status of each panel conversion.
7. Examine the panels and diagrams of the converted function panels.
8. Close all the scope VIs that were created.

End of Exercise 8-10

H. Using LabVIEW with HiQ

National Instruments produces HiQ, a high-performance numerical analysis application for Windows 98/95/NT. HiQ is an interactive problem-solving environment to organize, visualize, and document real-world math, science, and engineering problems. This package solves real-world problems using a methodology that combines a notebook interface, interactive analysis, data visualization, an extensive math library, and a programming language called HiQ-Script.

HiQ has advanced data visualization capabilities, such as 3D graphs, that enable you to easily interpret data generated from within the HiQ environment or other applications. With this feature, you can use HiQ with other test and measurement packages (such as LabVIEW) to enhance your overall application.

To ease the implementation of HiQ in your application, the software package includes features to import and export data files in a variety of formats, including delimited text or binary data. Because of these features, you can develop LabVIEW applications that write test data to a text or binary file and then import that data into HiQ for additional analysis or visualization.

In addition to writing data files for HiQ, LabVIEW also includes a series of VIs that enable you to transfer data between these two applications in various formats. You can also control HiQ from LabVIEW, launching HiQ, opening and closing HiQ notebooks, and running HiQ scripts. With these features, you can create applications that automatically transfer the data to a HiQ notebook for analysis by an end user.

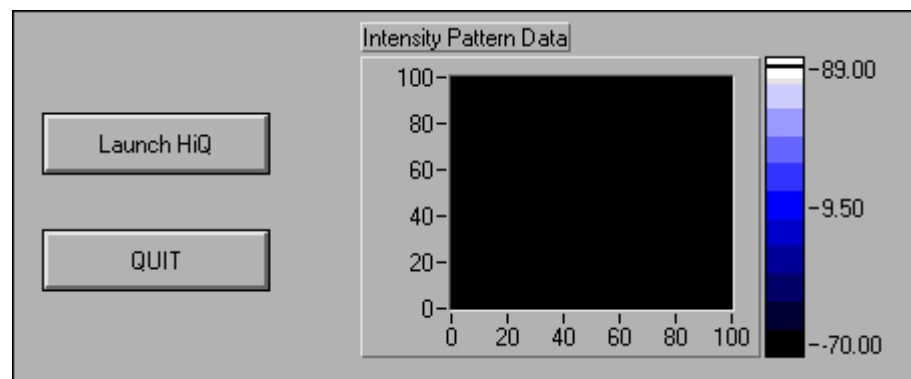
Exercise 8-11(Optional—Windows only) Data to HiQ.vi

Objective: To transfer test data between LabVIEW and HiQ.

There are several instances where you may need to view data in three dimensions. For example, you may need to determine the temperature gradients along a surface, the contour plot of a stretch of terrain, or even the changes in a signal's frequency over time. While such a set of 3D data can be plotted within LabVIEW, the intensity charts and graphs within LabVIEW basically display 3D data as a series of intensities (colors) on a 2D graph. For true 3D graphing of the data, an analysis package such as HiQ proves useful.

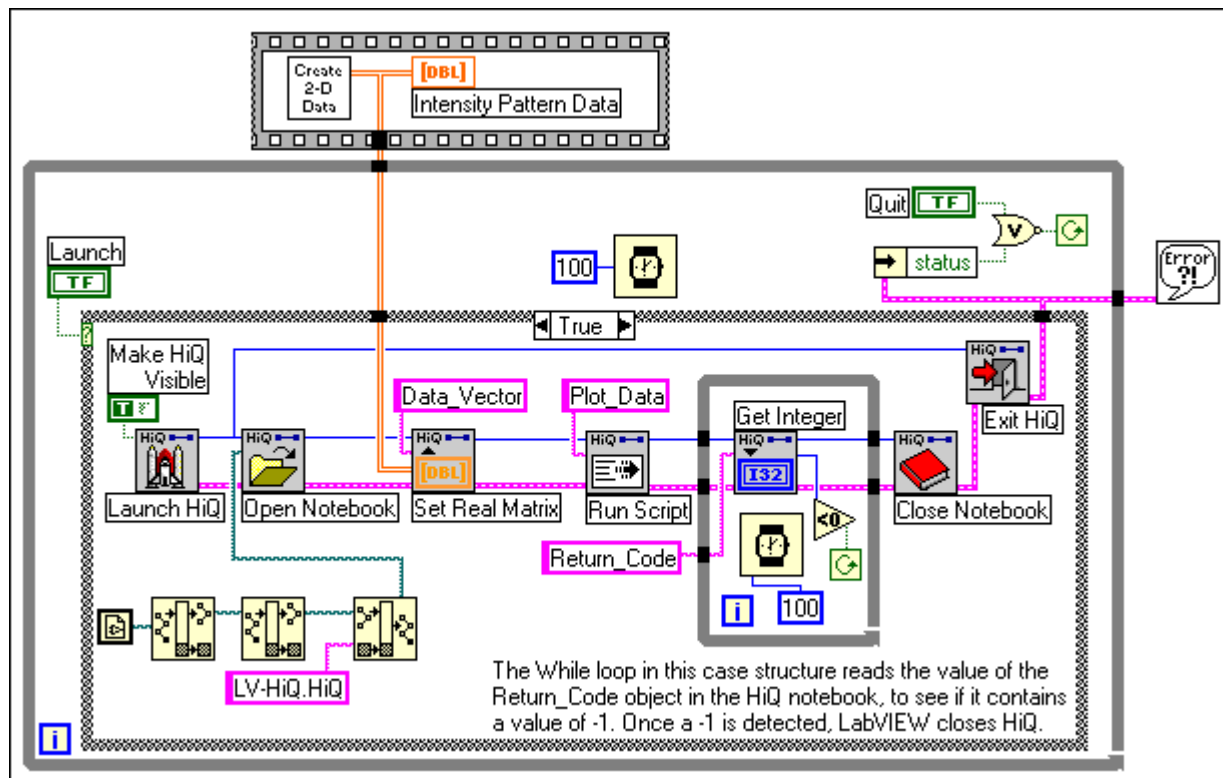
In this exercise, you will examine a LabVIEW application that generates a set of test data, loads a notebook into HiQ programmatically, and ports the data to HiQ for plotting. This LabVIEW application waits until it receives a proper return code from HiQ before closing the HiQ notebook. This exercise shows how you can seamlessly launch HiQ from a LabVIEW VI and transfer data between the two applications.

Front Panel



1. Open the **Data to HiQ** VI in Basics2.llb. This VI has already been built.
2. Show the block diagram.

Block Diagram



1. Examine the block diagram. When examining the VIs in the diagram, use the LabVIEW Help Window to provide more information on each VIs inputs and outputs.

The sequence structure at the top of the diagram creates a set of decaying sinusoidal data using the **Create Intensity Data** subVI and plots it on the VI's intensity graph. Once the data is plotted, the While loop begins to execute.

Whenever the Launch HiQ button is pressed, a series of VIs are executed to communicate with HiQ. All of these VIs communicate to HiQ using OLE (Object Linking and Embedding):



Launch HiQ VI (Communication»HiQ subpalette). This VI launches the HiQ application. The Make HiQ Visible Boolean instructs LabVIEW to show HiQ's front panel when it is launched.



Open Notebook VI (Communication»HiQ subpalette). This VI loads a notebook into the HiQ application. The notebook for this exercise, LV-HiQ.HiQ, has already been built, and should be in the same directory as Basics2.11b so that the notebook can be found.

The HiQ notebook contains several objects that will pass data between HiQ and LabVIEW. First, the notebook contains a 2D array of double-precision numbers (known in HiQ as a Real Matrix) called

Data_Vector, which will accept the intensity data created in LabVIEW. The notebook also contains a script (program) called Plot_Data, which will plot the data in the Data_Vector object into a 3D graph in HiQ. Finally, the notebook contains an I32 (integer) value called Return_Code, which will be set to a predetermined value in HiQ whenever LabVIEW is to close the notebook.



Set Real Matrix VI (Communication»HiQ subpalette). This VI takes the 2D array of double-precision values from the sequence structure and sends them to the Data_Vector object in the HiQ notebook.



Run Script VI (Communication»HiQ subpalette). This VI runs a script located in an open HiQ notebook. In this case, the script that is run is Plot_Data, which will take the data now in the HiQ Data_Vector object and plot it on a 3D graph.



Get Integer VI (Communication»HiQ subpalette). The **Get Integer** VI retrieves an integer value from the Return_Code object in the HiQ notebook. This value is checked in a loop; if the value is non-negative, the loop stops executing. Until that happens, this integer value is checked every 100 ms.

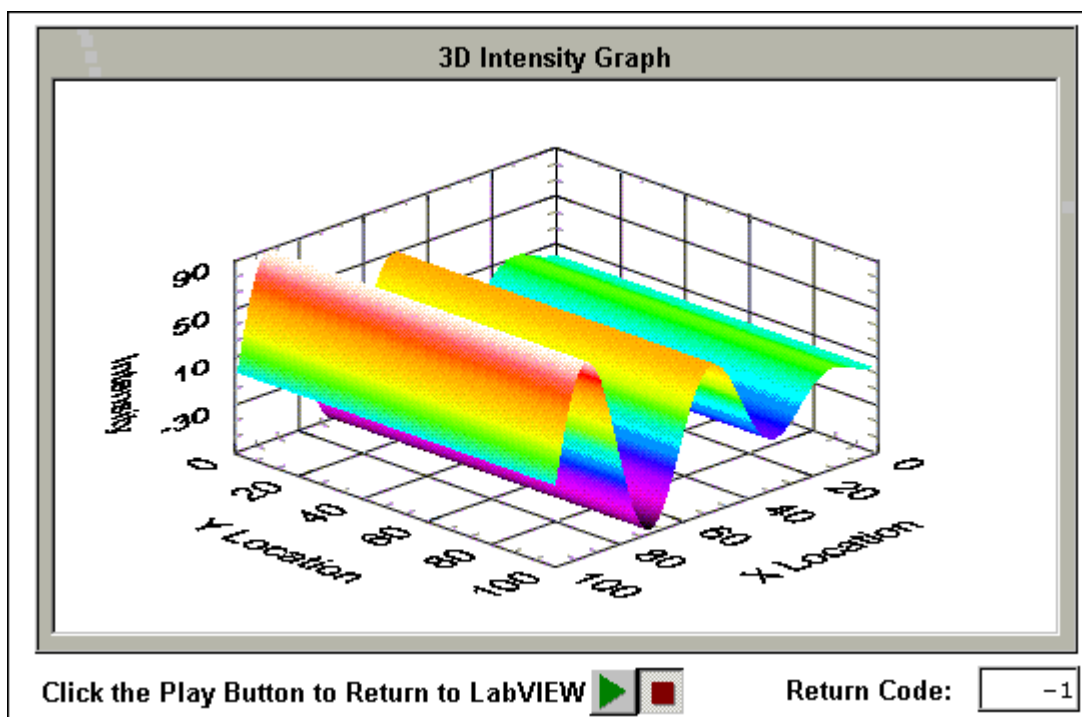


Close Notebook VI (Communication»HiQ subpalette). Once the integer value from Return_Code is non-negative, the While loop stops executing. The next function in line, **Close Notebook**, unloads the LV-HiQ notebook from HiQ.



Exit HiQ VI (Communication»HiQ subpalette). When run, this subVI closes a HiQ program currently loaded into memory.

2. Switch to the front panel and run this VI. Once an intensity graph appears, press **Launch HiQ** to execute the data transfer between HiQ. HiQ will launch, the LV-HiQ notebook will be loaded, and the data created in LabVIEW will be plotted on a 3D graph. You may want to maximize the HiQ window and notebook at this point, which should appear similar to the one shown on the following page.



3. Examine the HiQ notebook, using the HiQ scrollbars to look at various parts of the notebook workspace. At the top of the notebook is the Intensity Graph data, which has been transferred from LabVIEW and plotted. Under the graph is the Return_Code object, which currently contains a value of -1. As you scroll further down the notebook, you will see the raw data ported from LabVIEW (the Data_Vector object), as well as the script used to transfer the data to the 3D graph (Plot_Data).
4. You can view the 3D graph from different angles. Click and drag your mouse cursor over the 3D graph and you will rotate the graph accordingly. If you hold down <Ctrl> while click-dragging the mouse cursor up and down, you can also zoom in and out of the graph.
5. Once finished, press the Play button on the HiQ worksheet. The Return_Code object will momentarily contain a value of zero, and then LabVIEW will close the HiQ notebook and application. If you want, you may press **Launch HiQ** again from the LabVIEW VI to repeat the process.
6. With HiQ closed, press QUIT on the **Data to HiQ** VI to stop the application. Close the VI without saving any changes.



Play button

End of Exercise 8-11

Summary, Tips, and Tricks

- LabVIEW allows you to import images from popular graphic formats onto your VI's front panel. LabVIEW also has decorations and attribute nodes available to enhance front panels.
- You can customize front panel controls and indicators using the Control Editor. A Type Definition is a custom control in which all instances of the custom control have the same data type, but the appearance of the control or indicator can be customized individually. A Strict Type Definition is a custom control in which the data type and appearance of the control is fixed, and changing the "master" changes all instances of the control.
- Picture controls display the picture data type. You develop pictures using an extensive library of drawing functions or by modifying the examples.
- 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.
- Occurrences can be used to control separate, synchronous activities. In particular, you use these functions when you want one part of a block diagram to wait until another part of a block diagram finishes a task without forcing LabVIEW to poll.
- Each copy of a reentrant VI in memory has its own data space. Calls to reentrant VIs execute independently, without danger of sharing data between copies of the VI. If you plan to call a VI that uses an uninitialized shift register from more than one place in your application, you should make that VI reentrant.
- You can use the CVI Function Panel Converter to convert CVI function panels into LabVIEW VIs. This tool is extremely helpful when you need instrument drivers that may be available in CVI, but not in LabVIEW.

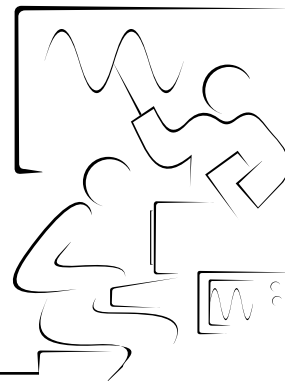
Additional Exercises

- 8-12 Open **Moving Truck.vi**. Modify the VI so the truck appears to move across the road. Use the Control Editor to replace the slide pointer with the truck graphic and the slide groove with the road graphic. Save your VI when you are finished.

Notes

Notes

Appendix



This appendix contains the following sections of useful information for LabVIEW users:

- A. ASCII Character Code Equivalents Table
- B. Additional Information

A. 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
53	123	83	S

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
73	163	115	s

Hex	Octal	Decimal	ASCII
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
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

B. Additional Information

Application Notes

Many LabVIEW application notes are available. You can request these notes from National Instruments or access them from the LabVIEW bulletin board. Although some application notes may pertain to LabVIEW for Macintosh, they also apply to LabVIEW for Windows. Contact National Instruments for more information about application notes.

Instrument Driver Library

The LabVIEW Instrument Driver Library contains VIs for more than 600 GPIB, serial, and VXIbus instruments. These drivers come with the LabVIEW installation package. You can also request these instrument drivers from National Instruments or access them from the LabVIEW web site at <http://www.natinst.com>.

Notes

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: LabVIEW Basics II Course Manual

Edition Date: February 1999

Part Number: 320629F-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.

Thank you for your help.

Name

Title

Company

Address

E-mail Address

Phone (

)

 Fax (

)

Mail to: Technical Publications
National Instruments Corporation
11500 North MoPac Expressway
Austin, Texas 78759-3504

Fax to: Technical Publications
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)

- | | | | |
|--|---|---|--|
| ☐ Automotive | ☐ Industrial systems -
factory floor/integrator | ☐ Pharmaceutical | ☐ Test, measurement,
and instrumentation |
| ☐ Computer | ☐ Medical | ☐ Aero/avionics | ☐ Telecommunications |
| ☐ Consumer products | ☐ Military/space | ☐ Semiconductor | ☐ University/education |
| ☐ Electronics | ☐ Paper/pulp | ☐ ATE/automated test | |
| ☐ Graphics | ☐ Petrochemical/plastics | ☐ Other _____ | |

If you are currently a customer of National Instruments, please check the products you use:

- | | | | |
|--|--|--------------------------------|---|
| ☐ LabVIEW™ | ☐ HiQ™ | ☐ DAQ | ☐ Fieldbus™ |
| ☐ LabWindows/CVI™ | ☐ ComponentWorks™ | ☐ SCXI™ | ☐ IMAQ™ Vision |
| ☐ BridgeVIEW™ | ☐ VirtualBench™ | ☐ GPIB | ☐ Serial |
| ☐ Lookout™ | ☐ Measure™ | ☐ VXI | ☐ Motion control |

Please check the operating system(s) you use:

- | | | | |
|-------------------------------------|--------------------------------------|--|--------------------------------|
| ☐ Windows 95 | ☐ Windows 3.1 | ☐ Mac OS | ☐ HP-UX |
| ☐ Windows NT | ☐ Sun | ☐ Concurrent PowerMax | |

Please check the bus architecture(s) you use:

- | | | | |
|-----------------------------------|------------------------------|------------------------------------|------------------------------|
| ☐ PC/XT/AT | ☐ PCI | ☐ Macintosh | ☐ DEC |
| ☐ PCMCIA | ☐ VME | | |

What other products are of interest to you:

- | | | | |
|---|---|-------------------------------|---|
| ☐ LabVIEW | ☐ HiQ | ☐ DAQ | ☐ Fieldbus |
| ☐ LabWindows/CVI | ☐ ComponentWorks | ☐ SCXI | ☐ IMAQ Vision |
| ☐ BridgeVIEW | ☐ VirtualBench | ☐ GPIB | ☐ Serial |
| ☐ Lookout | ☐ Measure | ☐ VXI | ☐ 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 use a PC regularly in my instrumentation system. |
| ☐ I influence product purchases. | ☐ I develop virtual instrumentation applications. |
| ☐ I evaluate and recommend software. | |

Which statement best describes your function in the company? (check only one)

- | | | | |
|---|--|---|--|
| ☐ Education | ☐ Calibration | ☐ Government/legal | ☐ Production test |
| ☐ Manufacturing/
automation | ☐ Engineering
management | ☐ Research/R&D/grad
student | ☐ Systems integrator/
hardware |
| ☐ Reseller/sales | ☐ Purchasing/contracts | ☐ Software developer | ☐ Software consultant |
| ☐ Service/repair | ☐ Student/co-op | ☐ Design | ☐ 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

- | | |
|--|---|
| ☐ <i>Measurement and Automation Catalogue</i> | ☐ <i>Automation View™</i> newsletter |
| ☐ <i>VXI Product Solutions Guide</i> | ☐ Third-Party Solution CD |
| ☐ <i>Academic Catalogue</i> | ☐ <i>NI News</i> e-mail newsletter |
| ☐ <i>Instrumentation Newsletter™</i> | |

Industry-specific Literature

- | | | | |
|---|--|---|--|
| ☐ Aerospace | ☐ Semiconductor | ☐ Analytical chemistry | ☐ Industrial automation |
| ☐ Telecommunications | ☐ Vibration/acoustics | ☐ Education | ☐ Laboratory automation |
| ☐ Automotive | ☐ Physiology | ☐ Test and measurement | |

Product Literature (check up to three)

- | | | | |
|---|--|--|--|
| ☐ LabWindows/CVI | ☐ Analysis | ☐ GPIB | ☐ PXI™ |
| ☐ ComponentWorks | ☐ HIQ | ☐ GPIB chip kit | ☐ IMAQ |
| ☐ TestStand™ | ☐ Measure | ☐ HS488™ | ☐ Customer education |
| ☐ LabVIEW Add-On
Toolkit pack | ☐ LabVIEW
productivity study | ☐ Virtual instrumentation
software | ☐ Computer-based
instruments |
| ☐ BridgeVIEW | ☐ DAQ | ☐ VXI | ☐ SCXI signal
conditioning |
| ☐ Lookout | ☐ Low-cost DAQ | ☐ VME | |

Additional Literature

- | | |
|---|---|
| ☐ NI Global Services | ☐ LabVIEW Technical Resource Subscription Card |
| ☐ Customer Education Course Schedule | |



11500 North Mopac Expressway Austin, TX 78759-3504
Tel: 512-794-0100, (800) 433-3488 • Fax: 512-683-9300
info@natinst.com • www.natinst.com